Best Practices for the Deployment and Management of Production HPC Clusters

Robert McLay, Karl W. Schulz, William L. Barth, and Tommy Minyard
Texas Advanced Computing Center (TACC)
The University of Texas at Austin
{mclay,karl,bbarth,minyard}@tacc.utexas.edu

ABSTRACT

Commodity-based Linux HPC clusters dominate the scientific computing landscape in both academia and industry ranging from small research clusters to petascale supercomputers supporting thousands of users. To support broad user communities and manage a user-friendly environment, end-user sites must combine a range of low-level system software with multiple compiler chains, support libraries, and a suite of 3rd party applications. In addition, large systems require bare metal provisioning and a flexible software management strategy to maintain consistency and upgradeability across thousands of compute nodes. This report documents a Linux operating system framework, (*LosF*), which has evolved over the last seven years to provide an integrated strategy for the deployment of multiple HPC systems at the Texas Advanced Computing Center. Documented within this effort is the high-level cluster configuration options and definitions, bare-metal provisioning, hierarchical HPC software stack design, package-management, user environment management tools, user account synchronization, and local customization configurations.

1. INTRODUCTION

Commodity-based Linux clusters have arisen as the dominant compute platform for high-performance, scientific computing based on price-performance considerations, the continued maturity and adoption of the Linux operating system, the availability of low-latency, high-speed interconnects for clusters, and the rapid processor improvements brought about by competition in the x86 processor segment. This increased cluster deployment trend is seen over a range of deployment sizes starting with small, individual researcher clusters to very large-scale multi-user systems which dominate the Top500 list that tracks LINPACK performance for the world's fastest supercomputers. Indeed, on the June 2011 list, 92% of the submissions are Linux based with 82% of the submissions designated as clusters [20].

The Texas Advanced Computing Center (TACC) at The University of Texas at Austin has a history of deploying leading-edge Linux clusters to support thousands of researchers from a diverse range of scientific disciplines. These include both general-purpose supercomputers like *Ranger* [12] (deployed in 2008 with 62K cores) and *Lonestar* [7] (deployed in 2011 with 22K cores) along with specialized resources like the *Longhorn* remote visualization cluster [8] (deployed in 2010 with 512 NVIDIA GPUs). Since these systems support a wide range of users, it is paramount to design and maintain a user-friendly environment which is flexible enough to provide a range of necessary low-level HPC software (e.g. InfiniBand drivers, multiple compiler chains and MPI stacks, parallel file systems, custom Linux kernels) and desired application support libraries (e.g. I/O, linear algebra, FFT, performance profiling). In tandem, a unified management strategy is required which can maintain consistency and upgradability across thousands of individual compute nodes and support servers with minimal system administration overhead.

In this report, we outline the overall strategies adopted for the deployment and management of production Linux clusters at TACC based on the evolution over seven years to devise a Linux operating system framework (*LosF*) which provides a single, integrated approach for managing hierarchical software stacks on multiple HPC and visualization resources. The resulting approach is based on the combination of both in-house developed software and available open-source tools. Included in this discussion is the overall design/cluster management philosophy and high-level cluster configuration options (Section 2), the bare-metal provisioning strategy (Section 3), the hierarchical HPC software stack design (Section 4), a newly developed user-environment management tool named `Lmod` (Section 5), and the package-management system implementation and build system for various third party packages installed on TACC resources (Section 6).

2. LINUX OPERATING SYSTEM FRAMEWORK

While the majority of components required to deploy a stable, secure, and mature HPC cluster of modest size are readily available, in practice we see example clusters from all sectors (academia, industry, and government) which suffer from poor cluster management and misconfiguration. In part, this is due to a lack of qualified professionals with the expertise to sufficiently architect and debug the complex requirements associated with HPC clusters that necessitate the integration of software and hardware from a number of different vendors. Clusters also require very pro-active patch strategies in order to maintain security and take advantage of performance gains garnered from a rapidly evolving list of HPC software. While a number of open-source and commercial toolkits exist to aid sites in performing bare-metal provisioning to deploy an HPC system (e.g. Rocks [26], Oscar [11], UniCluster [14], Bright Cluster Manager [1], and

Copyright is held by the author/owner(s)
SC '11, November 12-18, 2011, Seattle, Washington USA
Copyright 2011 ACM 978-1-4503-0771-0/11/11.

others), the primary focus of many of these efforts is to provide turn-key solutions during the hardware deployment phase. Substantially less effort is given towards providing a flexible management ecosystem over the 3-5 year lifespan of typical HPC clusters, particularly in the open-source arena. Consequently, this opens the door for mismanagement and leads some frustrated users and institutions to assume that clusters are not as reliable and productive as their "big-iron" counterparts. In reality, of course, a properly configured and managed cluster can in fact be very stable and provide significant productivity to its users.

2.1 HPC Cluster Management Philosophy

The overall HPC cluster management philosophy and development of *LosF* which has evolved over the last seven years is premised on the following desired features and design criteria. In particular, we desire:

- a single, integrated management strategy which can be applied to multiple systems, each with a range of underlying support servers (e.g. compute nodes, login nodes, resource managers, parallel filesystem servers)
- an end-to-end approach for system management ranging from initial bare-metal provisioning through to cluster retirement
- simple and customizable deployment tools which can adapt to the needs of new vendor hardware offerings quickly
- simple application and support library software state synchronization mechanisms with low overhead; support for scalable syncing across thousands of servers; *an incremental sync mechanism is preferred over repeated bare-metal provisioning*
- flexibility to easily customize provisioning mechanisms
- simple and scalable configuration/flat file controls
- traceability across cluster state changes (e.g. historical records for configuration file changes)
- flexible user-environment management and software usage support
- minimal system administration overhead with sufficient traceability and robustness for one or more administrators to manage multiple large systems simultaneously
- to leverage open-source tools where possible

To fulfill these desires, *LosF* has been assembled using a number of available open-source tools combined with in-house developed software to provide a unifying cluster management system for all production clusters at TACC in order to meet the outlined goals above. For the internally developed tools, the efforts are primarily categorized into two broad efforts. The first effort is focused on software management and synchronization and serves as the unifying mechanism for high-level cluster management integrating provisioning and long-term software update/configuration processes. The developed tools to support this cluster management effort are predominantly written in Perl and shell scripts which provide familiar constructs for system administrators who are the primary consumers. The second development effort is the introduction of a new user-environment management tool (`Lmod`) which provides a convenient, hierarchical approach for users to access local HPC software. This tool is written in Lua and is discussed in detail beginning in Section 5. In addition to these development efforts, a unified effort to build and maintain specialized 3rd party packages in a hierarchical manner using the RPM package management system is discussed in Section 6.

```
# $Id: config.machines 750 2011-05-03 12:06:13Z karl $

#--------------------------------------------------
# Cluster Definitions
#
# Define a name for each cluster you wish to manage.
#--------------------------------------------------

[Cluster-Names]

clusters = Longhorn Ranger Lonestar

#--------------------------------------------------
# Node Types/Cluster Config
#
# Delineate the domain name and different types of
# nodes in each cluster.  Note that the software on
# each node type can be managed separately.
#
# Regular expression patterns can be used to map
# individual hostnames to a give node type.
#--------------------------------------------------

[Lonestar]

domainname = ls4|ls4.tacc.utexas.edu

master     = master
nfs        = home1|data1|exp1
admin      = admin1
login      = login[1-4]
oss        = oss[1-9]+
mds        = mds[1-4]
bigmem     = c300-10[1-5]
gpgpu      = c300-20[1-8]
compute    = c3[0-4][0-9]-[1-3][0-2][0-9]
build      = build
gridftp    = gridftp[1-2]
```

Figure 1: Example portion of top-level *LosF* config file (*config.machines*) defining clusters and nodal appliances.

2.2 Cluster/Node Designations

Keeping with the desire to have simple configuration options, *LosF* adopts a keyword-driven input file to control configuration options for one or more defined clusters. Similar in style to other provisioning toolkits (e.g. Rocks), *LosF* adopts an appliance based approach in which individual servers are grouped together into appliance groups which provide similar functionality and, therefore require a similar set of base operating system (OS) packages, configurations, and 3rd-party packages. In order to define individual clusters and appliance types which belong to each cluster, a top-level configuration file is defined in which local site administrators identify desired settings. As an example, Figure 1 presents portions of the top-level configuration file (*config.machines*) used at TACC which defines three individual production clusters (*Longhorn, Ranger,* and *Lonestar*).

From this example, we see that for each defined cluster, a separate input block can be provided to define all the various node types (or appliances) which are present in the cluster. For the *Lonestar* cluster example shown in Figure 1, there are eleven node types defined and the general form of the definition is to supply a $key = value$ pair where *key* is the node type designation and *value* contains the individual hostnames which belong to the named node type. Note that

for convenience, the configuration file parser supports regular expression syntax such that a range of hostnames can be easily defined within a single node group. As an example, the login node definition entry of `login = login[1-4]` for the Lonestar cluster indicates that hosts *login1*, *login2*, *login3*, and *login4* are the only four hosts which are defined as login nodes. Note that in practice, we choose locally to define hostnames based on server functionality. For example, nodes which are used to define object-storage servers (oss) and and meta-data servers (mds) for Lustre parallel file systems receive hostnames which reflect this functionality ($mds1, oss1$, etc). In addition to defining desired node types for each cluster, another necessary input is the network domainname for each cluster. This is required to allow *LosF* to differentiate between clusters and for the customization of software packages and configurations on a per-cluster basis.

2.3 Software/Configuration Synchronization

The cluster/node definitions outlined in the previous section are used by *LosF* to organize which OS and 3rd-party software packages are installed or are updated on individual cluster hosts. As will be discussed further in Section 6, we have adopted the RPM package management system as the basis for *all* software installations. Note that the desired RPMs to be installed on a node have the potential to come from several distinct sources: (1) a released Linux distribution (e.g. RHEL or CentOS), (2) an external or mirrored repository providing updates for Linux distribution packages, and (3) locally-built packages maintained by site staff to provide the customized HPC environment required for a diverse range of users. In addition to controlling specific package installations, the cluster/node definitions are also used by *LosF* to synchronize configuration files (or any other desired ascii file) across all hosts.

The primary tool used to perform software and configuration management is an *LosF* utility named `update` which is responsible for bringing any cluster node into a known software state and configuration. `update` can be executed locally on an individual node or in parallel across an entire cluster to verify the state of individual hosts and/or push out configuration/package updates based on local administrator changes. Note that past experience deploying large clusters has shown that incremental updates are more scalable than bare-metal reprovisioning of modified OS images and hence `update` was designed to detect node status and bring it to a known good state (including the addition and removal of software) along with configuration file management (including user credentials).

To configure various file synchronizations, *LosF* queries an additional cluster-specific input file using the cluster definitions named in the global *config.machines* configuration. An example portion of the input file for the *Lonestar* cluster referenced previously is shown in Figure 2 which illustrates representative files that are locally managed along with example configuration syntax. In the first `[ConfigFiles]` section, we see several examples of common files that system administrators need to customize based on local hardware configurations and desired user policies (e.g. *fstab* for controlling locally mounted file systems, *access.conf* for defining login control policies, *motd* for customizing system-wide announcements, etc). Synchronization for the named files occurs via `update` by comparing the contents of the locally

```
# $Id: config.Lonestar 764 2011-06-24 17:12:07Z karl $

#-----------------------------------------------
# Configuration File Syncing
#-----------------------------------------------

[ConfigFiles]
/etc/motd                    = yes
/etc/fstab                   = yes
/etc/exports                 = yes
/etc/pam.d/sshd              = yes
/etc/security/access.conf    = yes
/etc/security/limits.conf    = yes
/etc/sysconfig/network       = partial
...

-----------------------------------------------
# Runlevel Services Syncing
#-----------------------------------------------

[Services]

xfs                          = off
autofs                       = off
ntpd                         = on
sendmail                     = off
cups                         = off
limic                        = on
...
```

Figure 2: Example portion of *LosF* cluster-specific input file (*config.Lonestar*) detailing configuration file options.

installed file to a reference version which is managed by local site administrators. Normally, the reference versions are defined on a per node-type basis and are stored in a shared file system accessible across the cluster. In these cases, reference files are organized in the top-level *LosF* config directory as follows:

`$LOSF_CONFIG/const_files/[Cluster Name]/[Node Type]`

As an example, to manage the access.conf file for login nodes on our Lonestar cluster, we customize a reference file at:

`$LOSF_CONFIG/const_files/Lonestar/login/access.conf`

and the contents of this reference file are updated and verified on each login node during an `update` process. Note that if an administrator inadvertently modifies the contents of a locally-installed file, the differences are cached during the `update` process such that the modifications can be examined and applied to the production reference files during a subsequent `update` process if desired. While this syncing process provides a very convenient mechanism to ensure consistency across a specific node type, our experience has shown that in certain circumstances, additional control is required to fine-tune one or more hosts belonging to a defined node group. For example in the *Lonestar* cluster configuration, we recall that four login nodes are defined but it is conceivable that a site might want to restrict login access to one of these nodes (e.g. for staff use only). Other than the login access differences, the nodes are to remain identical and as opposed to requiring the definition of a new *staff-login* node type, *LosF* allows administrators to override the default node configuration with a host-specific reference file. For this *Lonestar* example, if a `login/access.conf.login4` file is defined, then `update` will synchronize this file on the *login4* host, while using the default `login/access.conf` as

```
NETWORKING=yes
NETWORKING_IPV6=yes
HOSTNAME=c399-101.ls4.tacc.utexas.edu
#-------------------------------------begin-sync-losf-
#
# LosF partially synced file - do not edit entries
# between the begin/end sync delimiters or you may
# lose the contents during the next synchronization
# process. Knock yourself out adding customizations
# to the rest of the file as anything outside of the
# delimited section will be preserved.
#
# $Id: network 606 2011-01-27 17:19:31Z karl $

GATEWAY=149.76.4.1
#-------------------------------------end-sync-losf-
```

Figure 3: Example *network* configuration file which contains a partially synchronized file contents managed by *LosF*.

the basis to synchronize all remaining login node hosts.

In addition to illustrating some example configuration file paths, Figure 2 also indicates that two general synchronization possibilities are supported. In particular, both *full* and *partial* file synchronization can be configured. The full case simply means that the entire contents of the named file are to be managed by *LosF*. The partial designation indicates that only a portion of the named file is to be synchronized. This feature requires a begin/end delimiter (normally embedded as comments within the particular configuration file of interest) to be added within the synchronization file and *LosF* will then verify consistency solely within the delimited portion of the file allowing for non-common or host-specific settings to be defined within the rest of the file. As an example, Figure 3 shows the contents of a resulting `/etc/sysconfig/network` file that was generated for a specific compute node on the *Lonestar* cluster. The top contents of this file are created during the initial kickstart-provisioning mechanism (discussed further in Section 3) and because the specific hostname is embedded, the files cannot be identical across all compute node types. However, they can all share the same routing gateway and the partial synchronizing mechanism provides a customizable way to augment this file to accommodate local network configurations.

Another configuration option highlighted in Figure 2 is the capability to control various runtime services (controlled via the `[Services]` section). At TACC, our underlying Linux installations are derived from CentOS distributions which are not tuned by default for HPC installations. Consequently, we utilize this runlevel service configuration option to disable non-necessary services which can impact application performance (e.g. automounter, print daemons, X font servers, etc.) and to enable HPC-specific services like the LiMIC2 package which provides lightweight kernel-level primitives to optimize MPI intra-node communication on multi-core systems [25]. Note that several additional synchronization input options exist but are not included in Figure 2 for brevity. These include options to synchronize file-level permissions and to control the existence and mapping of any symbolic file links on individual cluster servers.

In practice, to provide further traceability across the cluster management system, we choose to keep all of the relevant *LosF* input files and corresponding reference synchronization files defined for each cluster node type under the control of a software configuration management (SCM) system. Locally, we use subversion for this purpose, a popular open-source version control system, but this could easily be substituted with other available SCMs. Consequently, traceability across last edit and authorship is visible among the various configuration file examples shown in in Figures 1-3.

In terms of software package synchronization, recall that we utilize the RPM system to maintain all package revisions. To support the maintenance of packages which fall out of the purview of a Linux distro, `update` provides a mechanism to synchronize 3rd party packages, custom kernels and GRUB configurations on a per-cluster and per node-type basis. For each cluster designation, an additional input file is allowed (e.g. *update.Lonestar*) which provides the flexibility to define additional RPM packages to be installed, removed, or upgraded across individual host resources. The hierarchical approach adopted for the creation and maintenance of local RPM packages is presented in more detail in Section 6, but once created, the location of locally created RPM binaries to be synchronized are exposed to *LosF* via an input control. In practice, we maintain all local RPMs in one or more shared-file systems, although the packages can also be installed over the network using the *http* protocol.

Interestingly, the motivation to support RPM synchronization from multiply defined shared-file systems locations is based on scalability concerns and the availability of file systems at different points during a node's life cycle. For example, once an HPC cluster has been provisioned it often includes a parallel filesystem to provide a scalable I/O resource for end-user applications (e.g. Lustre or GPFS). These filesystems are designed to support concurrent transactions across hundreds or thousands of clients where a traditional single network file server running NFS would be easily overwhelmed. Consequently, the parallel filesystems can also be leveraged to support cluster-wide system updates in a scalable fashion. In practice, we leverage this capability extensively to update thousands of nodes simultaneously when the parallel filesystems are available. When not available (e.g. during a bare-metal provisioning process), the `update` mechanism can revert to performing synchronization via a traditional filesystem.

A final responsibility for `update` is the verification of installed Linux OS updates which are released incrementally by distro providers. As we are primarily CentOS/RHEL based, we adopt the standard *yum* tool to perform distro-provided OS updates based on local repository mirrors in order to keep systems patched with community releases.

3. BARE-METAL PROVISIONING

An important consideration in the overall management of any production cluster is the choice of provisioning mechanism used to deploy OS images and ideally, customized 3rd party packages starting from bare-metal. Obviously at the scale required to support thousands of servers, the method must be automated and sufficiently scalable to perform multiple installations simultaneously. In previous deployments, we have utilized the NPACI ROCKs toolkit to perform provisioning which is based on a modified kickstart mechanism to deploy custom images over PXE from a defined master server. In more recent deployments, we have abandoned this approach in favor of *Cobbler* [2], an alternative open-source package designed for rapid setup of network installation environments. Both of these toolkits have similar management utility functionalities which allow administrators

to add/delete node definitions and assign network settings, along with providing server-side software support to extend DHCP, DNS, and TFTP services to registered hosts in order to support kickstart provisioning use PXE.

Unlike ROCKS, Cobbler does not provide the direct OS images that are to be provisioned and is designed as a light weight application with simple configuration mechanisms to provide very flexible customization options. Consequently, Cobbler-based provisioning begins first by importing a desired Linux distribution and then exposes simple command-line tools to define kickstart rules to be used during provisioning. After importing one or more distribution images, the next requirement is to define relevant kickstart files which control the base OS installation choices. An attractive feature of this approach is that site administrators have convenient control over the kickstart files in order to customize installations for alternate hardware configurations, disk partitioning schemes, raid configurations, and more. Another particularly attractive feature of Cobbler is the ability to simply define *snippets* which are common blocks of code that can be run at pre-configured times during the provisioning process. This allows Cobbler to be trivially integrated with *LosF* in that a snippet can simply instantiate `update` during the bare-metal provisioning process. Consequently, newly provisioned nodes automatically receive the latest software stack revision along with all desired configuration file settings.

Given the scale of large HPC systems which can easily grow to a hundred or more compute racks, provisioning the entire system at once is normally not required. However, support is desired to install significant portions of a system simultaneously and Cobbler's PXE-based provisioning mechanism has been shown to be sufficient for large-scale HPC systems. As an example of the provisioning times encountered using Cobbler within *LosF* during TACC's most recent *Lonestar* deployment, the wall clock time required to provision 8 racks (or 384 servers) simultaneously was approximately 30 minutes. Note that a non-trivial portion of this deployment measurement includes the time to complete BIOS initialization, memory scanning, and Linux kernel boots into the final desired node configuration.

4. HPC SOFTWARE HIERARCHY

Users of HPC resources demand the highest levels of performance. Additionally, they often work with specialized software applications with particular software requirements. As a result, it is necessary for a resource provider to support a number of different compilers, MPI distributions, support libraries, and community applications. Due to the difference in C and Fortran function calling conventions and incompatibilities between C++ ABIs, it is necessary to build a matrix of supported software. Applications that link with pre-built libraries must generally use the same compiler toolset as that used for the library itself. This leads to a combinatorial increase in deployed packages and libraries. For each supported compiler, there must be a build of each supported serial library and MPI stack, and under each compiler/MPI combination, there must be a build of each supported parallel library.

In addition, software packages are updated regularly to provide access to new versions containing bug fixes and new features. With each new version added, there is often the requirement to support the older versions as well in order to maintain continuity for the existing user community. In light of these considerations, maintaining the entire HPC software stack for a production resource is a challenging problem.

4.1 Directory Structure for Local Software

In order to clearly separate and maintain the numerous versions of libraries and applications on the HPC resources at TACC, we place all local software in a directory structure relative to a single top-level path, usually `/opt/apps`. Our naming scheme is as follows:

1. Applications are stored as `/opt/apps/`*Application_Name/Application_Version*.
   For example, the software version control program `git` version 1.7 would be located in `/opt/apps/git/1.7`.

2. Compiler-dependent libraries are stored under a compiler and version directory `/opt/apps/`*Compiler-version/Package_Name/Package_Version*.
   For example, the `C++ boost` library version 1.3 built with `gcc` version 4.2 would be found under `/opt/apps/gcc4_2/boost/1.3`.

3. Compiler- and MPI-dependent parallel libraries must be stored with both compiler and MPI directory encoding `/opt/apps/`*Compiler-version/MPI-version/Package_Name/Package_Version*.
   For example, the parallel solver package PETSc 3.1 built with `gcc` version 4.2 and the MPI stack `mvapich` version 1.2 would be found under `/opt/apps/gcc4_2/mvapich1_2/petsc/3.1`

It is clear from this layout that each new version of a package has a unique place in our optional software directory tree with no chance of collision between versions. For each compiler version there can be multiple versions of each library. For the parallel libraries, there is a multiplicative effect; i.e., for two compilers and three MPI stacks there could be up to six versions of a given library. This directory structure makes it straightforward to support multiple compilers with multiple, dependent MPI implementations simultaneously.

4.2 Environment Modules

Given the large number of installed optional software packages, there is a clear need for a software system that supports the discovery of such packages and management of the shell environment enabling their use. Environment Modules [17, 22, 23] are a long-standing solution to this problem. Under the Environment Modules system, users are presented with the `module` command for finding available packages on the system and for importing them into their UNIX shell environments. The Environment Modules system uses a collection of "module files" to maintain the necessary path, library path, usage instructions, and other information associated with each package.

A user wishing to use a particular package "loads" a module file through the `module` command. The module file contains commands that can load other modules or change the user's environment such as adding a directory to the user's `PATH` or `LD_LIBRARY_PATH` or setting other shell environment variables. If a user "unloads" a loaded module then all the

additions to the user's shell environment are reversed. All previously added directories are removed from the user's PATH, and the other shell variables are unset. For each package installed on the system, there is an associated module file. Additionally, the module command translates the module file directives to be appropriate for the user's chosen UNIX shell.

Another important feature of the module system is that a package can be referenced via a name and version, which allows users access to more than one version of a package. If users load a package by name without the version then they get the default as specified by the systems administrators. Alternatively, a user may select a particular version by issuing "`module load Foo/1.1`" to load version 1.1 whereas "`module load Foo`" will load the default, say version 1.0. Letting users control what software they use and optionally, which version, is key to providing a friendly and flexible environment. Users who want the latest beta version of a package can be accommodated while also maintaining older, potentially more stable revisions.

Module files typically add to the user's PATH and define other necessary environment variables. At TACC, our library module files also provide a standardized naming scheme for environment variables which name the header file and library directories. For the FFTW2 package, for example, we provide the environment variables TACC_FFTW2_INC to point to the header file directory and TACC_FFTW2_LIB for the library directory. In this way users' build tools can use these variables instead of hard-coding the paths to particular a version of the library. This approach is particularly advantageous for end users so they can access new software updates easily within their build system.

The module system was first described in 1991 [22] and in subsequent papers [23, 27]. There is also a website for the TCL-based version of Environment Modules [17]. Many users of HPC systems were first exposed to modules in the mid-1990s on Cray systems that used them [23]. Note that the module system available from [17] has all of the features that are described above, however, it does not have built-in support for managing the hierarchical software matrix across compilers, MPI stacks, and other parallel libraries described previously.

4.3 Module Hierarchy

With a large matrix of compilers, MPI stacks, and package versions available, the choices facing system administrators for the deployment of module files are important. Consider the following example system which has two compilers installed, GCC (version 4.5) [5] and Intel (version 11.1) [13] two MPI implementations MVAPICH (version 1.2) [10] and OpenMPI (version 1.5) [18], and a user community that desires the parallel linear algebra library PETSc (version 4.1) [16]. There will be four different versions of the PETSc installation for the four different pairings of compilers and MPI stacks. One strategy is to have a flat naming scheme for the four module files for PETSc:

1. PETSc-4.1-mvapich-1.2-gcc-4.5
2. PETSc-4.1-mvapich-1.2-intel-11.1
3. PETSc-4.1-openmpi-1.5-gcc-4.5
4. PETSc-4.1-openmpi-1.5-intel-11.1

Unfortunately, there are specific potential problems with adopting a flat scheme. Users are always presented with the four versions of the PETSc module files required to support the four compiler/MPI combinations. If another version of PETSc is added (say, PETSc v4.2), there will be four new module files added to the list. If we were to add in all the packages that an HPC system provides, it would be difficult for users to find the packages they wish to use. For example, there are more than 700 modules supported on Ranger at TACC. Furthermore, the onus for ensuring package compatibility is placed directly on the user.

In order to partially mitigate this problem, we have created a hierarchy of environment module files. Our strategy is to place the module files along side the package installations themselves and to use the MODULEPATH environment variable to ensure that only appropriate module files are visible to users while keeping incompatible files hidden.

The module command reads an environment variable MODULEPATH which specifies a colon-separated list of directories containing module files. Users can only load module files that are in their MODULEPATH. Initially on our systems the MODULEPATH only contains /opt/apps/modulefiles. When a compiler module is loaded, the module system prepends a directory to MODULEPATH that contains the module files for packages built with that particular compiler. For example, if the intel/11.1 module is loaded, the module system would prepend /opt/apps/intel11_1/modulefiles to MODULEPATH.

At this point, the user will then be able to see packages built with the Intel 11.1 compiler. More importantly for a parallel computing environment, this will also make visible all of the MPI stacks built against the Intel 11.1 compiler (e.g. MVAPICH2 1.6 and OpenMPI 1.5 above). If the user then loads the mvapich2/1.6 module, /opt/apps/intel11_1/mvapich2_1_6/modulefiles will be prepended to MODULEPATH, and parallel libraries based on MVAPICH2 1.6 and Intel 11.1 will then be available.

At this point, only one PETSc module appears in the users list of available modules instead of the 4 modules in the previous flat module scheme. By only making appropriate modules visible, users cannot load a mismatched module at the outset when setting up their environments. However, without further improvements to the module system, it is still possible for a user to leave the wrong module in their environment when switching between compilers or MPI stacks.

5. Lmod: A NEW ENVIRONMENT MODULE SYSTEM

At TACC, we originally deployed the TCL-C module system [17] using the hierarchical module directory structure described in the previous section. The above layout of module files works well for users until they try to change compilers or MPI stacks. Without some support from the module system, users can easily find themselves with mismatched modules. For example, if a user starts with the Intel compiler and related MVAPICH modules loaded, and swaps the Intel compiler module for the GCC compiler module, the compiler and MPI environment will be mismatched. To address this problem, a new implementation of the module system, Lmod [6], was developed and made available as an open source project under the MIT license.

The name Lmod was chosen because this tool for managing users' environment is a complete rewrite of the module system using a language call *Lua* [9,24]. *Lua* is a simple yet

powerful scripting language. Among *Lua*'s many strengths are two features found useful here. The first is that functions are first class objects which means that functions can be handled like variables. This greatly simplifies the code for loading and unloading of module files. The second feature is that the main *Lua* data structure is a table which stores both array elements and hash tables in a clear way. `Lmod` supports reading of TCL module files from the TCL-C module system, so there is no immediate need for administrators to translate the module files into *Lua* when migrating from TCL-C modules to `Lmod`.

`Lmod` has several key improvements over other module system. First, `Lmod` tracks changes to `MODULEPATH`. When it changes state, `Lmod` unloads any modules which are no longer in the `MODULEPATH`. It then tries to reload any modulefiles it can with the new `MODULEPATH`. Any modules that can not be loaded are saved in an inactive state. Turning back to the example from the previous section, PETSc version 4.1 may not be available with the GCC 4.5 compiler and the OpenMPI version 1.5 MPI stack. However, when `MODULEPATH` changes again, `Lmod` tries to reload the inactive modules. When `Lmod` cannot reload a previously loaded module, it warns the user that this step was unsuccessful.

Now when a user changes compiler and/or MPI stacks, the dependent modules change automatically. It is all handled internally by `Lmod` through the tracking of the state of `MODULEPATH`.

5.1 Important Lmod: Features

`Lmod` provides several important features that protect users from making potential mistakes. The most important feature is reloading the dependent modules when their dependencies are changed. Another important feature is that users cannot load two versions of the same module.

`Lmod` maintains several important pieces of state information about a user's current shell environment, including: which modules are active or inactive and where their associated module files are located in the filesystem. This data is stored in a single *Lua* table across several environment variables. In ASCII form, this string contains single and double quotes and can be quite long, typically between 4000 and 6000 characters long. The length will depend on how many modules the user has loaded. To avoid shell limitations on environment variable length, and to simplify the management of shell quoting rules, we convert the long ASCII text string into a uuencoded [3] string which contains no special characters and then split that string into a series of environment variables each 512 bytes long. Every time the module command is run, it reads the uuencoded chunks from the environment variables, concatenates them, and uudecodes the full string back to an ASCII text version of the table that can be easily evaluated by *Lua*. This extra state information stored by `Lmod` is used to support the automatic loading and unloading of modules when the `MODULEPATH` changes.

In order to protect users from inappropriate loading of similar modules, `Lmod` prevents users from loading two versions of the same module. If a user tries to load one version of a module and then another, the first module is unloaded and the second module is loaded in its place. For example, executing `module load Foo/1.1; module load Foo/1.2` loads package `Foo` version 1.1, then Foo v1.1 is unloaded, and `Foo` version 1.2 is loaded. Additionally, we have extended the module file language to include a new command: `family`. This command takes a single string as an argument, and `Lmod` prevents more than one module from the same family from being loaded at the same time. For example, all TACC compiler module files have set their family to "compiler", and all the MPI module files set the family name "MPI". Users with one compiler module already loaded will see an error message when they try to load a second compiler without unloading the first. This is similarly true when users attempt to load a second MPI module file. For the very small percentage of users who need multiple modules from the same family loaded at the same time, they may set the `LMOD_EXPERT` environment variable to bypass this restriction.

The use of a hierarchy does simplify users search for modules to load as they only see modules that are appropriate for their current compiler-MPI pairing. This is important on Ranger where we have over 700 total module files. The listing of available modules contains less than 300 entries which can be listed in columns that fit into one or two screen outputs. This makes for easy scanning of available modules.

Due to the potential for explosive growth in the number of packages built and installed on TACC systems, not all packages are supported under every compiler and MPI combination. Additionally, the "`module avail`" command only finds modules available under the current `MODULEPATH`, which is volatile under the module hierarchy described above and likely to change with a user's changing module environment. These two factors conspire to make it difficult for users to discover all of the packages available system-wide. Therefore, we have added an additional module command, "`module spider`" that searches all parts of the module hierarchy for available module files. Because providing detailed information about each package would generate many pages of output, this command has three modes. The first, "`module spider`", with no additional arguments, provides a concise list of modules and a short description of each package. In the second mode "`module spider modulename`", lists all the versions of all module files that match modulename. Finally "`module spider module/version`" gives all the information that the module file has and lists what compiler and MPI stack that will provide that version. Both commands take simple regular expressions to aid in locating desired modules.

TACC provides users with a default shell environment that includes a minimum set of modules to get users started. Many users will want other modules loaded as well every time they log in. The first option is place module commands in their shell start-up files (e.g. ∼/.bashrc, ∼/.cshrc). There are however some issues with this method that will be discussed in the next section.

`Lmod` also provides a second option. Users can log in and issue module commands to load and unload module files to build their desired environment. Once satisfied with the list of modules loaded, they can issue "`module setdefault default`" which caches the current module state into a file in `∼/.lmod.d/`. This default state will loaded instead of the standard list of modules at login. Additionally, users can create other named lists of modules to be loaded. Only the `default` list will be loaded during login but users can create other lists via "`module setdefault` *foo*" and load that named set with "`module getdefault` *foo*".

5.2 Shell Startup Issues

Working with shell startup mechanisms presents two particular difficulties. The first is primarily a problem specific to HPC systems. Users who place module commands in their startup files can cause problems on parallel filesystems when their jobs are run on a large number of processors due to excessive file operations on their home directories during the SSH phase of MPI job startup. Users may wish to place module command in their ∼/.bashrc or ∼/.cshrc files, knowing that when setup correctly, those files will be sourced in interactive and login shells. Care must be taken if users run on a large number of processor cores where each core invokes a shell and each shell is trying to read through module files. As a result, we recommend that users utilize the `setdefault` technique or that they wrap their module commands in their startup files. The relevant syntax for bash users at TACC is as follows:

```
if [ -z "$_BASHRC" -a "$ENVIRONMENT" != BATCH ]; then
  export _BASHRC="read"
  module load git fftw2
fi
```

On our systems we set the variable `ENVIRONMENT` to `BATCH` on the compute nodes. Since the environment variables of the user's submission shell are propagated to the compute nodes in their jobs, modules loaded in the submission environment will also be present in their batch job shell environment.

The second issue relates to the bash shell. Bash may or may not source a system bashrc file during an interactive non-login shell startup. We want the module command to be available for interactive as well as login shells so we have rebuilt bash to always read a system bashrc. When sourced, it loads our initial startup scripts that are in `/etc/profile.d/*.sh`. This is where the module command is defined and where we provide an initial set of modules to load. In order for the `setdefault` command to work, the system startup scripts issue:

```
module getdefault default || module load TACC
```

This way the user's default is loaded if it exists, otherwise the system defaults stored in the `TACC` module file are loaded.

Bash is also different from other shells such as Tcsh and Zsh in that there are no system startup files read during the startup of a bash shell script. Instead the value of `BASH_ENV` is used as a file to be sourced. In our case, we have it point to the file which defines the `module` command. We define `BASH_ENV` for all the shells we support. This way any user executing a bash shell script will have the `module` command defined in their bash script, an important consideration for jobs running under a batch scheduling environment.

5.3 Recording Software Usage with Lmod

Another attractive benefit of `Lmod` is the ability to track loaded modules during a shell logout. This provides usage information to help prioritize our support efforts by delineating between frequently used packages versus those with little or no usage. Also, in some cases a new package is installed that duplicates the functionality of an old package, but includes new features or enhanced performance. Having access to historical usage data allows us to quickly identify which users might be interested in trying the new package.

```
...
%define APPS /opt/apps
%define MODULES modulefiles
%include compiler-defines.inc
%include mpi-defines.inc
%define INSTALL_DIR %{APPS}/%{comp_fam_ver}/
                    %{mpi_fam_ver}/%{name}/%{version}
%define MODULE_DIR  %{APPS}/%{comp_fam_ver}/
                    %{mpi_fam_ver}/%{MODULES}/%{name}
...
```

Figure 4: Inclusion of parameter variable checking in FFTW2 RPM spec file.

6. PACKAGE MANAGERS

TACC systems use the RedHat-derived CentOS Linux distribution for their underlying operating systems, so it was a natural choice to leverage the existing RPM [4, 21] environment to handle specialized HPC software built and installed by TACC. The use of a package manager provides us with several benefits.

- It simplifies installation of software on the nodes.
- It captures institutional knowledge on how to build and rebuild packages allowing multiple staff members to maintain a particular package.
- The specification file contains the instructions to build the software and it also generates the associated environment module file.
- When the software is installed via *LosF*, the new module file is installed simultaneously. This means that users can access this software immediately with no manual intervention by a system administrator required to describe the software.
- Removing the RPM package file removes all the software and the module file that accesses it. No user can "load" a module file for non-existent software because both have been removed.

RPMs are created through a plain-text specification file (i.e. the "RPM spec file") which describes what source files to use, how to unpackage and build them (if necessary), what files to include in the final RPM, and where to install those files. For each optional software library or application TACC installs, we create a single, parameterized RPM spec file which is used to generate carefully named and crafted RPMs for each compiler-MPI pair we support. The compiler and MPI implementation names and versions are encoded into the name of the RPM in order to make these RPMs distinct within the RPM database.

A full sample RPM spec for the FFTW2 library [15] file is shown in Figure 11.There are several important additions to a typical RPM spec file layout that we include to allow it to be used to generate multiple differently-named RPMs for each compiler-MPI pair. Note that RPM spec files are compiled into binary RPM files with the `rpmbuild` command. Usually this is done by specifying `rpmbuild -bb fftw2.spec`, but in order to pass the compiler and MPI parameters into the compilation environment, the local software maintainer(s) must define two additional parameters, using the `-D` command-line option, e.g.:

```
rpmbuild -bb -D 'is_intel11 1' -D 'is_mvapich2 1'
  fftw2.spec
```

These variables are utilized via two include files we embed into each spec file: `compiler-defines.inc` and `mpi-defines.inc`, see Figure 4 for an example. These in turn are

used to define the RPM spec file variables, INSTALL_DIR and MODULE_DIR which control the installation and module file paths and the name of the eventual RPM. Sample contents for these files can be found in Figures 5 and 6. In this example, we demonstrate cases covering use of compilers from the Intel 11.x [13], PGI 10.x [19], and GCC 4.4.x [5] families, and either MVAPICH2 1.6 [10] or OpenMPI 1.3.3 [18].

```
% define comp_fam error

% if "%{is_intel11}" == "1"
  % define comp_fam intel
  % define comp_fam_ver intel11_1
  % define comp_fam_name Intel
% endif

% if "%{is_pgi10}" == "1"
  % define comp_fam pgi
  % define comp_fam_ver pgi10
  % define comp_fam_name PGI
% endif

% if "%{is_gcc44}" == "1"
  % define comp_fam gcc
  % define comp_fam_ver gcc4_4
  % define comp_fam_name GNU
% endif

% if "%{comp_fam}" == "error"
  % {error: Compiler not defined}
  exit
% endif
```

Figure 5: compiler-defines.inc

```
% define mpi_fam error

% if "%{is_mvapich2}" == "1"
  % define mpi_fam mvapich2
  % define mpi_fam_ver mvapich2_1_6
% endif

% if "%{is_openmpi}" == "1"
  % define mpi_fam openmpi
  % define mpi_fam_ver openmpi_1_3_3
% endif

% if "%{mpi_fam}" == "error"
  % {error: MPI not defined}
  exit
% endif
```

Figure 6: mpi-defines.inc

Next, as shown in Figure 11, the %package and %description macros are called with the optional naming argument (-n) to tell RPM to generate an alternately named file. Then, the build process begins. Here, as shown in Figure 11, compiler-load.inc and mpi-load.inc are included to setup the correct compiler and MPI environment for the build. Based on the defined compiler and MPI, the correct environment modules are loaded, and the traditional autotools-style compiler variables are set (.e.g. CC, CXX, FC, etc).

After these variables are set, the MPI compiler variables can also be set and the configure script run. In this case, the INSTALL_DIR RPM variable created above is used to set the autotools prefix. After FFTW2 is configured, the rest of the build process is conducted.

Figure 9 shows how in the %install macro section of the spec file, we construct the contents of the FFTW2 environment module file–in this case written in Lua. This step leverages both the MODULE_DIR and INSTALL_DIR RPM variables to create the module file in the correct place and set the TACC_* environment variables correctly.

```
...
%package -n %{name}-%{comp_fam_ver}-
         %{mpi_fam_ver}
Summary: FFTW 2.x local binary install
Group: System Environment/Base
%description
%description -n %{name}-%{comp_fam_ver}-
              %{mpi_fam_ver}
FFTW2 RPM
...
```

Figure 7: Package and Description macros in FFTW2 RPM spec file.

```
...
%build
%include compiler-load.inc
%include mpi-load.inc
...
```

Figure 8: Build macro in the FFTW2 RPM spec file.

```
...
%install
## Module for fftw2
mkdir -p $RPM_BUILD_ROOT/%{MODULE_DIR}
cat > $RPM_BUILD_ROOT/%{MODULE_DIR}/%{version}.lua \
      << 'EOF'
local help_message = [[
The FFTW2 modulefile defines the following variables:
TACC_FFTW2_DIR, TACC_FFTW2_LIB, and TACC_FFTW2_INC
for the location of the FFTW %{version} distribution,
libraries, and include files, respectively.

To use the FFTW library, compile your code with:
        -I$TACC_FFTW2_INC
and add the following options to the link step:
        -L$TACC_FFTW2_LIB -lfftw
Version %{version}
]]
help(help_message,"\n")
local fftw_dir="%{INSTALL_DIR}"
setenv("TACC_FFTW2_DIR",fftw_dir)
setenv("TACC_FFTW2_LIB",pathJoin(fftw_dir,"lib"))
setenv("TACC_FFTW2_INC",pathJoin(fftw_dir,"include"))
EOF
...
```

Figure 9: Install macro in the FFTW2 RPM spec file.

In the final RPM macro section, as shown in Figure 11, %files, the name of the RPM file to create and the files to collect are described. The final RPM will be named %{name}-%{comp_fam_ver}-%{mpi_fam_ver}-%{version}-%{release}.rpm, or fftw2-intel11-mvapich2_1_6-2.1.5-1.rpm in this example. Running the full combination of three compilers and two MPI stacks would lead to six RPM files to install for this version of FFTW2. Depending on demand and disk space for installation, not all six of these combinations may be deployed on resources at TACC, but they are all accommodated from a single spec file.

7. CONCLUSIONS

This report highlights some of the techniques used to perform large-scale Linux HPC cluster management which have evolved over the past seven years at the Texas Advanced Computing Center. Included in the discussion is an overview

```
...
%files -n %{name}-%{comp_fam_ver}-%{mpi_fam_ver}
%defattr(755,root,install)
%{INSTALL_DIR}
%{MODULE_DIR}
...
```

Figure 10: Files macro in the FFTW2 RPM spec file.

of *LosF* which is used to provide a single, integrated approach for managing hierarchical software stacks on multiple HPC and visualization resources. Also included is the motivation for the newly developed `Lmod` user environment management system which provides a systematic way to expose a hierarchical HPC user stack to system users while also allowing local administrators to track software package usage. Finally, the approach taken to manage the build process and life-cycle of 3rd party packages using the RPM package management system was described including relevant spec file examples which illustrate package generation for multiple compiler and MPI tool-chains.

8. REFERENCES

[1] Bright cluster manager. `http://www.brightcomputing.com`.

[2] Cobbler. `http://fedorahosted.org/cobbler`.

[3] en.wikipedia.org/wiki/uuencoding. Wikipedia Article describing the uuencode technique.

[4] fedora.linuxsir.org/fedoradocs/rpm-guide/en/. RPM Guide.

[5] gcc.gnu.org. Gnu Compiler Collection.

[6] Lmod: A lua based environment module system. `http://lmod.sourcefourge.org`.

[7] Lonestar4 linux cluster. `http://www.tacc.utexas.edu/resources/hpc/#lonestar`.

[8] Longhorn visualization cluster. `http://www.tacc.utexas.edu/resources/visualization/#remote`.

[9] Lua programming language. `http://www.lua.org`.

[10] mvapich.cse.ohio-state.edu/overview/mvapich2/. MVAPICH/MVAPICH2 Project: MPI over InfiniBand, 10GigE/iWARP and RoCE.

[11] Open cluster group. oscar: A packaged cluster software stack for high performance computing. `http://www.openclustergroup.org`.

[12] Ranger linux cluster. `http://www.tacc.utexas.edu/resources/hpc/#constellation`.

[13] software.intel.com/en-us/articles/intel-compilers/. Intel Compiler Suite.

[14] Univa unicluster. `http://www.univa.com/products/unicluster.php`.

[15] www.fftw.org. FFTW website: Discrete Fourier Transforms.

[16] www.mcs.anl.gov/petsc/petsc-as/. PETSc: Portable, Extensible Toolkit for Scientific Computation.

[17] www.modules.org. Environment Modules website.

[18] www.open-mpi.org/. A High Performance Message Passing Library.

[19] www.pgroup.com. Portland Group Compilers.

[20] Top500 supercomputer sites. `http://top500.org`, June 2011.

[21] E. C. Bailey. *Maximum RPM*. SAMS Publishing, 1997.


```
%% FFTW2 SPEC File
Summary:   FFTW2
Name:      fftw2
Version:   2.1.5
Release:   1
License:   GPL
Vendor:    www.fftw.org
Group:     System Environment/Base
Source:    fftw-%{version}.tar.gz
Packager:  nobody@example.com
Buildroot: /var/tmp/%{name}-%{version}-buildroot

%define APPS /opt/apps
%define MODULES modulefiles
%include compiler-defines.inc
%include mpi-defines.inc
%define INSTALL_DIR %{APPS}/%{comp_fam_ver}/
                    %{mpi_fam_ver}/%{name}/%{version}
%define MODULE_DIR  %{APPS}/%{comp_fam_ver}/
                    %{mpi_fam_ver}/%{MODULES}/%{name}

%package -n %{name}-%{comp_fam_ver}-%{mpi_fam_ver}
Summary: FFTW 2.x local binary install
Group: System Environment/Base
%description
%description -n %{name}-%{comp_fam_ver}-%{mpi_fam_ver}
FFTW2 RPM

%prep
rm   -rf $RPM_BUILD_ROOT
mkdir -p $RPM_BUILD_ROOT/%{INSTALL_DIR}
%setup -n fftw-%{version}

%build
%include compiler-load.inc
%include mpi-load.inc
export MPICC=`which mpicc   || /bin/true`
export MPIF77=`which mpif77 || /bin/true`
./configure CFLAGS="-O2 " FFLAGS="-O2" --enable-mpi \
  --prefix=%{INSTALL_DIR} --enable-threads
make DESTDIR=$RPM_BUILD_ROOT install
make clean

%install
## Module for fftw2
mkdir -p $RPM_BUILD_ROOT/%{MODULE_DIR}
cat > $RPM_BUILD_ROOT/%{MODULE_DIR}/%{version}.lua \
      << 'EOF'
local help_message = [[
The FFTW2 modulefile defines the following variables:
TACC_FFTW2_DIR, TACC_FFTW2_LIB, and TACC_FFTW2_INC
for the location of the FFTW %{version} distribution,
libraries, and include files, respectively.

To use the FFTW library, compile your code with:
        -I$TACC_FFTW2_INC
and add the following options to the link step:
        -L$TACC_FFTW2_LIB -lfftw
Version %{version}
]]
help(help_message,"\n")
local fftw_dir="%{INSTALL_DIR}"
setenv("TACC_FFTW2_DIR",fftw_dir)
setenv("TACC_FFTW2_LIB",pathJoin(fftw_dir,"lib"))
setenv("TACC_FFTW2_INC",pathJoin(fftw_dir,"include"))
EOF

cat > $RPM_BUILD_ROOT/%{MODULE_DIR}/\
      .version.%{version} << 'EOF'
#%Module3.1.1#########################################
set     ModulesVersion      "%{version}"
EOF

%files -n %{name}-%{comp_fam_ver}-%{mpi_fam_ver}
%defattr(755,root,install)
%{INSTALL_DIR}
%{MODULE_DIR}
```

Figure 11: FFTW 2 RPM spec file (`fftw2.spec`)

[22] J. L. Furlani. Modules: Providing a flexible user environment. In *Proceedings of the Fifth Large Installation Systems Administration Conference (LISA V)*, pages 141–152, 1991.

[23] J. L. Furlani and P. W. Osel. Abstract yourself with modules. In *Proceedings of the Tenth Large Installation Systems Administration Conference (LISA '96)*, pages 193–204, 1996.

[24] R. Ierusalimschy. *Programming in Lua*. lua.org, 2006.

[25] H.-W. Jin, S. Sur, L. Chai, and D. K. Panda. Lightweight kernel-level primitives for high-performance mpi intra-node communication over multi-core systems. *2007 IEEE International Conference on Cluster Computing*, 2007.

[26] P. M. Papadopoulos, M. J. Katz, and G. Bruno. Npaci rocks: Tools and techniques for easily deploying manageable linux clusters. *Concurrency and Computation: Practice and Experience Special Issue: Cluster 2001*, 2001.

[27] E. Whitney and M. Sprague. Drag your design environment kicking and screaming into the 90's with modules! In *Synopsys Users' Group*, Boston, 2001.