Practical Support Solutions for a Workflow-Oriented Cray Environment

Adam G. Carlyle, Ross G. Miller, Dustin B. Leverman, William A. Renaud, Don E. Maxwell
National Center for Computational Sciences (NCCS)
Oak Ridge National Laboratory (ORNL)
Oak Ridge, Tennessee, USA
{*carlylag, rgmiller, leverman, brenaud, maxwellde*}*@ornl.gov*

***Abstract*—The National Climate-Computing Research Center (NCRC), a joint computing center between Oak Ridge National Laboratory (ORNL) and the National Oceanic and Atmospheric Administration (NOAA), employs integrated workflow software and data storage resources to enable production climate simulations on the Cray XT6/XE6 named "Gaea". The use of highly specialized workflow software and a necessary premium on data integrity together create a support environment with unique challenges. This paper details recent support efforts to improve the NCRC end-user experience and to safeguard the corresponding scientific workflow.**

Monitoring and reporting of disk usage on Lustre filesystems can be a resource-intensive task, and can affect meta-data performance if not done in a centralized and scalable way. *LustreDU* is a non-intrusive tool that was developed at ORNL to address this issue by providing an end-user utility that queries a database which is populated daily for reporting disk utilization on directories in the NCRC Lustre file systems.

The NCRC system is housed at ORNL, and has sets of geographically remote end-users at (3) separate sites, with a corresponding support staff team at each location. Conveying system status information to each remote center in a timely manner became important early into the project. The NCRC System Dashboard is a web interface and a set of corresponding system checks created by ORNL support staff to concisely and expediently inform those operational teams remote from the main data center of changes in system status.

Filesystem issues and outages cause disruption to the automated workflow employed by NCRC end-users. *Lustre-aware Moab* is our response to this issue. By integrating knowledge of the filesystem state into the system's job scheduler, the workflow can be paused when a file system issue is detected. When the issue is resolved, affected jobs can be rerun, effectively rolling back the workflow's progression to a valid state.

***Keywords*-Cray; Gaea; NCRC; ORNL; NOAA; Lustre; Moab;**

I. INTRODUCTION

NCRC, and its associated primary computational resource, Gaea, supports NOAA's research and development community across the country and stands as the first remote production computing facility in NOAA's history.[1] As such, Gaea must support end-users with unique usage patterns. Production climate simulations run on Gaea, yet post-processing runs and ultimate archival of datasets are run at the end-users' home research center(s).

To automate this process, end-users on Gaea employ workflow software that manages the execution of climate research simulations as well as the movement of data across multiple file systems. In more typical HPC usage patterns, the occasional failed job has little impact; on Gaea, a single failed job can interrupt the overall workflow since the execution of simulations is necessarily serial in nature. Individual job failures must all be investigated. The resource manager queue state must be maintained and reconstructed after system outages. Geographically remote support teams must be notified concisely of system status changes in real time to be able to correlate hardware issues to user impact. The nature of the usage pattern mandates a modified approach to support on Gaea.

II. LUSTREDU

To facilitate the enforcement of soft quotas on Gaea's Lustre[2] file systems, regular snapshots of file system utilization must be generated. *LustreDU*[3] is a previously non-production tool ORNL developed for this task, and put into production for the first time on Gaea. In general, monitoring and reporting of disk usage on Lustre filesystems is a resource-intensive task. Too many simultaneous `du` commands to the filesystem can overwhelm the meta-data server and can affect performance. Regular bookkeeping of the file system is important for a number of reporting tasks, but must be done in a centralized and scalable way. LustreDU was developed with this in mind. It provides an end-user utility which queries a frequently-updated database for the purpose of reporting disk utilization on directories in the NCRC Lustre file systems.

A. LustreDU Command Line Utility

To the end-user, LustreDU is a simple tool, invoked from the command line, that lists the aggregated size of Lustre directories passed to it as a command line option as in Figure 1:

```
[user@host] (/scratch) $ lustredu ./somedir
Last Collected Date        Size    File Count  Directory
2012-04-10 07:29:33     103.91 KB           1  ./somedir
```

Figure 1. Typical LustreDU Command Line Tool Invocation

The command line tool is comprised of a handful of C++ classes to handle command line options, backend database interaction, and directory permission verification pre-output. Unlike `du`, LustreDU places minimal load on the Lustre meta-data server, querying it only to check that the user has permissions to view the next directory size to be output. The actual data to be output is queried directly from a mySQL database that is updated regularly by the LustreDU main process threads.

B. *LustreDU Code Internals Overview*

Internally, the LustreDU system consists of two programs: a master program that handles the majority of the work and a daemon that runs on each Lustre Object Storage Server (OSS) and responds to queries for object sizes. This is a client-server architecture, but in reverse: there's one client and multiple servers. Neither the master program nor the daemons use the Lustre API, although they do get information from files in the /proc/fs/lustre hierarchy.

The LustreDU system doesn't store information about each individual file. That would take too much space and we don't need that level of detail. Instead, it stores information on a per-directory basis. Specifically, for each directory, it stores the UID and GID for that directory and the total number and total size of all files contained in the directory and its children. With the exception of the file size information, all of this data comes from the Lustre Metadata Server (MDS) and we actually get it by running Nick Cardo's `ne2scan` utility.[4]

The only data that isn't available on the MDS is the file size. Lustre doesn't store the size of any files. Instead, it stores the size of each individual object in a file and calculates the total file size as needed. Those object sizes are stored on the OSS.

In general, the master process reads the output of `ne2scan` line by line. From each line, it gets the full pathname, UID, GID and the ID's of the objects that make up that file. It then queries each Lustre Object Storage Target (OST) for the size of that object. When it has received all the object sizes, it calculates the total size of the file.

Of course, the detailed explanation is rather more complicated than this. The daemon is simpler than the master program, so we'll discuss it first.

C. *LustreDU Daemon Processes*

As previously stated, each daemon listens on a network port for object ID's, looks up the object in the filesystem and returns the size. Each OSS has multiple OST's, and requests for objects on any of them can arrive in any order. In order to allow for asynchronous operation, the server process creates one thread for each OST plus one thread for receiving network requests and a final thread for sending the network replies.

The basic data flow is shown in Figure 2 and described below:

1) A file size request is received from the network. The listener thread examines the request and determines which OST it is for. It places the request in a queue for that OST and increments a semaphore.
2) The thread that is waiting on that semaphore wakes up and decrements the semaphore. It then looks up the requested file size, places it on an output queue for the network send thread and increments that queue's semaphore.
3) The network send thread wakes up, decrements its semaphore, pops the size value off the queue and sends it back to the client.

Each OST thread uses the `libext2fs` API to open and navigate the OST as if it contains a regular filesystem (which, of course, it does). Lustre objects are just files in this filesystem, and finding one object's size is as simple as performing a `stat` operation on the file.

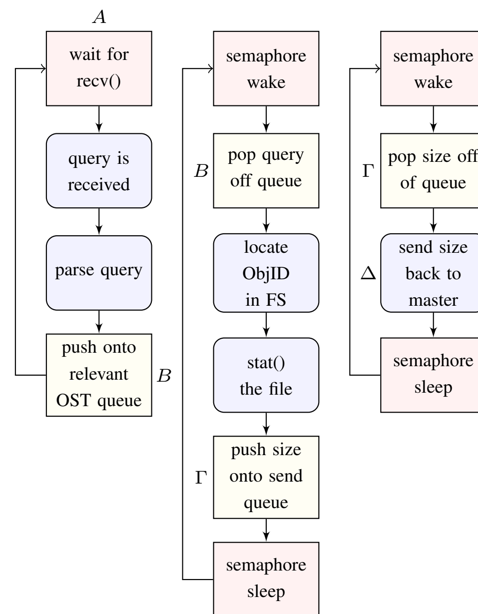

Figure 2. LustreDU OSS Daemon Threads Logic: Network Receive (left), OST Thread (center), Network Send (right)

D. *LustreDU Master Process*

The master program is a bit more complicated, mainly because there are far more threads and queues to manage. There are (2) threads for each OSS–(1) for network sends and (1) for receives–plus (1) thread for reading the `ne2scan` file and (1) for writing the output.[1] The system's master process logic is presented graphically in Figure 3,

[1]The largest filesystem at NCCS uses 96 OSS's which means the master program will use a total of 194 threads.

Figure 4, and Figure 5. In general, the process is the following:

1) The main thread reads a line of text from `ne2scan`. It parses the line for data on the file's name, UID, GID, mtime and the object IDs that make up the file. This parsed information is stored temporarily in an instance of a class called `FileObject`.
2) Each object ID is added to a request queue for the proper OSS along with a pointer to the associated `FileObject` instance and the semaphore for that queue is incremented.
3) The network send thread for that queue wakes up, decrements the semaphore, sends the request and goes back to sleep.
4) When the reply comes back, the receive thread (which has been blocked on the `recv` call), wakes up, retrieves the pointer to the `FileObject` and adds the size value from the reply to the total size value in the object.
5) If this is the last reply for this `FileObject`, then the pointer to the object is placed on a queue for completed requests.
6) When the `FileObject` appears on the completion queue, a final thread wakes up, parses the object and updates the table of directory information. After this, the `FileObject` is deleted.
7) When the main thread finishes parsing the `ne2scan` files, it waits for all other threads to finish their tasks. Once this happens, the main thread pushes all of the directory information up to a database.[2]

E. LustreDU Design Analysis

The reason the software uses all these threads, semaphores and queues is so that everything can run asynchronously. There is no way to predict which set of OST nodes any given file will use. Also, since this is running on a live filesystem, the OSS nodes are under load and the time required for one of them to respond to a size request can, and does, vary. It was considered unacceptable to hold up processing of files on other OSS nodes just because one OSS was slower to respond. The master program is therefore designed to read the `ne2scan` file as fast as possible in an effort to keep the network send queues full.

The same philosophy applies when data is received from the network. The order that the object size replies arrive in doesn't matter; if an OSS node or OST node happens to be slow to respond, other instances of `FileObject` can be processed by the completion thread. This architecture allows for fairly good load balancing without explicitly reordering or otherwise scheduling any of the network requests. In fact, on the filesystems at the NCCS, the main bottleneck seems to be the speed at which the `ne2scan` input file can be read.

[2]Writing the information out to a text file is also supported, but not used in production.

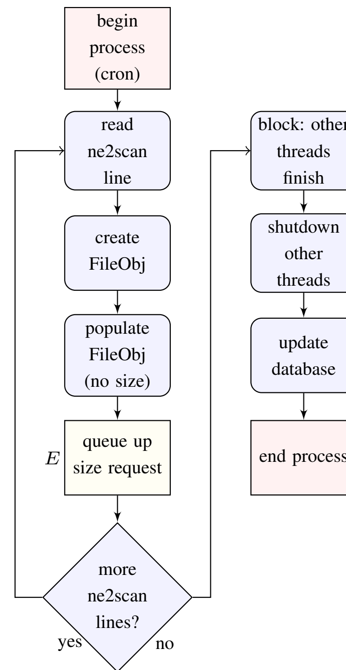

Figure 3. LustreDU Master Process Logic

As implied above, data are held in memory until all input has been processed. This is somewhat memory intensive, but necessary for decent performance. For each directory, the output will contain the total number and size for all files stored beneath that directory in the filesystem hierarchy. This means that for every file that is processed, the master program has to work its way back up the hierarchy, updating the size and file count for each directory as it goes. If this data were not kept in memory, the master program would have to perform a database update for each file it processes, effectively slowing down processing speed to the point of impracticality.

In order to ensure that the memory footprint of the process doesn't grow too large, each queue does have a maximum size. If a queue fills up, any thread that attempts to add another item will simply block until the queue has drained. The system is designed so that no single thread both adds and removes data from a particular queue, so deadlock is not an issue.

There are a number of implementation details that are not mentioned herein: discovering OSS IP addresses, mapping OSS node names to IDs, starting up and shutting down all threads, navigating the filesystem hierarchy on the OST nodes, and a description of the set of mutexes and semaphores that protect various data structures that are accessed by multiple threads. Though these details are important (and add significantly to the size of the codebase)

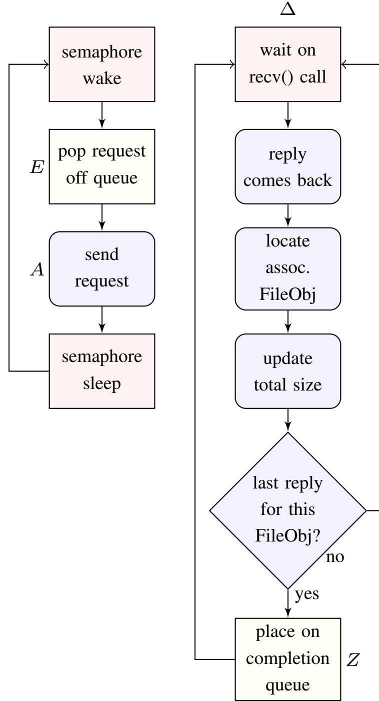

Figure 4. LustreDU Master Process Network Threads Logic: Send (left) and Receive (right)

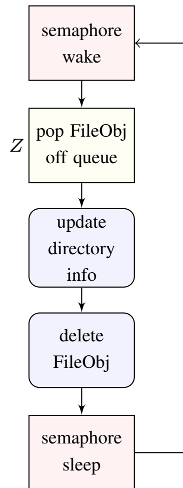

Figure 5. LustreDU Master Process Completion Thread Logic

they are beyond the scope of this discussion.

III. NCRC System Status

Geographically remote end-users and support staff are often interested in knowing the status of a system through some means other than direct access. An end-user or manager may need to know of system outages when direct system access is not possible. Or, an end-user may have experienced a login failure, and the root cause of the issue, whether local to one's site or due to an outage on the remote site, is unclear. Phone calls and emails can provide necessary answers, but these mechanisms may not provide sufficiently timely user feedback, particularly if the inquiry is made after normal business hours to a center that is not continuously staffed. Ideally, the end-user would have readily available system status information that is automatically updated.

NCRC employs the *Nagios*[5] software package to monitor many aspects of system status. While this information could conceivably be made available to users, it contains many additional tests that are of interest to system administrators, yet do not have a direct impact on system availability. Even tests of interest, such as response to `ping` commands, are broken down component-by-component. This has a number of drawbacks from an end-user perspective: it requires that end-users view many data points; it requires that end-users understand how various system components relate to one another; and it requires that end-users understand how these component interactions translate into overall system availability. The vast majority of end-users (and non-sysadmins, for that matter) do not have this level of insight.

Fortunately, it is a relatively straightforward task to analyze these variables programmatically, and to provide end-users with a status dashboard summarizing overall system availability in a succinct way. NCRC provides such a dashboard on NCRC webpage at: *http://www.ncrc.gov/dashboard*. Geographically remote support staff team members use the dashboard as a remote monitoring tool to correlate system outages with issue reports coming from their particular site's end-users.

A. The Main Script

A perl-based status script is the main driver of the system's logic and consists of several parts. The main driver program is a Perl script named `getstat.pl`. This script uses a collection of Perl modules, one per system being tested. The system module files provide routines to list all hosts that make up the system as well as routines to determine system status and return that information to `getstat.pl`. This does mean that as the number of systems tested grows we have an ever-increasing number of module files. This can easily lead to `getstat.pl` becoming littered with calls to each individual system. This is mitigated by an additional middleware module named `Downtime.pm`. This module provides the same routines as the system-specific modules but takes one additional parameter, *system name*, which it uses to call the appropriate back-end module. Thus, `getstat.pl` simply loops through a list of system names repeatedly calling routines in `Downtime.pm`. This provides much more concise code than calling individual system-specific modules.

After determining the current status of each system, one must determine if the state has changed. The status script makes use of an SQLite3 database to store states. This is done in lieu of a simple state file in order to

maintain an archive of system state changes. This becomes especially useful if we need to perform any troubleshooting post-incident. The table stores tuples of *system*, *state*, and *timestamp*, so determining the currently displayed state for a system is a trivial SQL `SELECT` operation. If the state as determined by the current iteration of the script doesn't match the most recent state in the database, a state change has occurred; we push the new information onto the database.

The ultimate job of the status script is to provide a summary of states in a way that can be easily processed by a website. At the time of this writing, the mechanism by which the website parses the data is being changed. The older method worked by consulting a different file for each system, obtaining the file's timestamp for the last state change, and comparing that file's MD5 checksum against known checksums for each state to place it concisely in time. This is being replaced by a much more straightforward method. The status script will write a single file containing *system*, *state*, *timestamp* tuples that will be parsed by the website and then displayed. Once the site has determined the state of a system (regardless of the method it uses), it applies a CSS tag appropriate for the detected state.

B. Dealing With False Positives

There are times when the Nagios tests (on which the script relies) return false positives. Typically, these are corrected the next time the script runs. In the absence of logic to handle this situation, this would have the unfortunate side effect of displaying an incorrect time for the state change, *i.e.*, a state change would be reported, although one never truly happened. This occurrence, dubbed a *flip-flop* is alleviated in the script. Each system has an assigned flip-flop threshold (in seconds). Any state change from one state to another state and then back to the original state within that threshold is deemed a false positive. When the script updates the states for display on the dashboard, this error is corrected.

The state of a system component at any given time can be *up*, *down*, or *degraded* within the context of the System Status code. To properly perform flip-flop detection, we must know about (3) such states; The current state (`n`), the previous state (`n-1`), and the state previous to that (`n-2`). Since we maintain a state archive, including timestamps, in the database, this amounts to a straightforward query. Flip-flop detection becomes a simple operation of comparing the state (`n`) to (`n-2`), and if they match, checking if the difference between the (2) timestamps is within the flip-flop threshold. If this is the case, state (`n-1`) is dismissed as a false positive.

C. Direct Staff Notifications

As the script runs, it generates a log detailing each iteration: which systems were checked, which ones were degraded and/or down, and the reasons any individual components were determined to be down. It also generates other output data (e.g., lists of state changes) and error data (e.g. execution failed for some reason). These `STDOUT` and `STDERR` data are then sent to staff so they can easily stay apprised of the changes that are being made without having to constantly monitor the status website.

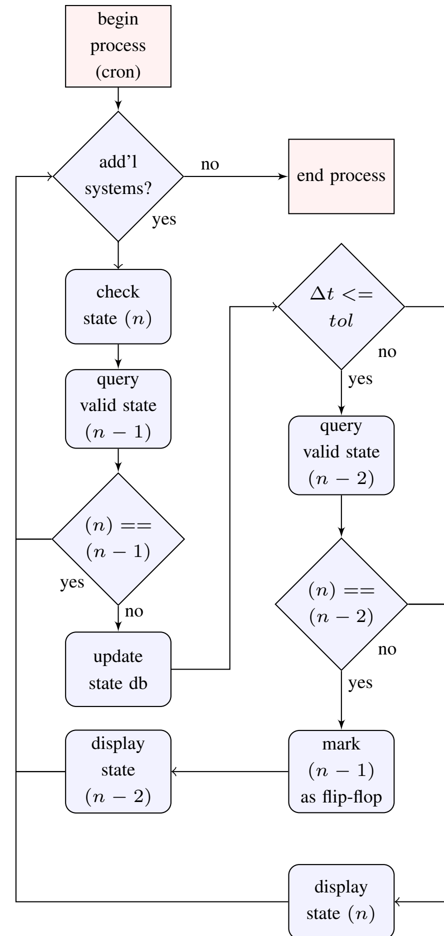

Figure 6. NCRC System Status Logic

IV. Lustre-Aware Moab

Unscheduled file system outages can be a particularly frustrating scenario for end-users of HPC resources. In the event of an unscheduled file system outage, the following series of events has been known to occur:

1) User submits a job to the batch scheduling system, where it waits, often for non-trivial amounts of time.
2) At some point prior to job execution, a shared file system on which a job depends goes down.
3) The job dies immediately upon starting, due to lack of appropriate underlying file system.
4) Compute resources that would have been in use by the job are quickly freed.

5) Another job from the queue starts on the freed compute resources, and also immediately dies.
6) The problem escalates until the batch scheduling queue is manually paused my an admin.

To address this issue, we have developed a way of programmatically creating a reservation for specific file system resources in Moab in the event that a file system becomes degraded or goes offline.

A. Lustre File System Checks

Detecting issues with a Lustre file system can be a challenge. It is difficult to detect problems in general, but even when problems are successfully detected, discerning whether or not the issue warrants a work stoppage is a similarly difficult exercise. When a file system issue does warrant a work stoppage, batch scheduling manager (Moab) reservations are typically created on any computational resource effected by the offending file system; this prevents any new jobs from the queue from starting execution. The primary piece of monitoring software that we use to detect issues with the file systems is *Nagios*, which allows us to write custom plugins and event handlers to detect and react to system problems.

A Nagios plugin is basically a script or binary that, when executed on the Nagios server, causes another script or binary to be executed on a Nagios client. There are a few transmission protocols available for this purpose; we employ the Simple Network Management Protocol (SNMP) herein. The client-side Nagios script, in turn, returns some piece of useful information to the server. This information includes a return code corresponding to a component state (*ok*, *warning*, *critical*, or *unknown*) and a description that yields more detail about the exact state of the check being queried.

A Nagios event handler is a script that is executed in the event of a monitored component state change. Event handlers can be programmed to react differently if we are in a *soft* or *hard* error state. File system issues that return non-ok Nagios checks initially cause a soft error state. This continues until Nagios returns a number of consecutive non-*ok* statuses that is greater than some threshold (`max_check_attempts`) set by the admin. Once the number of consecutive non-*ok* checks surpasses the threshold, the check enters a hard error state.

B. Nagios Check Plugins

The Lustre-aware Moab system uses (2) plugin-based checks to assess file system health. One check examines a configuration file which contains a list of: (a) all Lustre block devices in the given file system and, (b) the corresponding systems on which those devices should be mounted. This check effectively ensures that the appropriate block devices are mounted on the appropriate computational systems.

The other check simply looks at `/proc/fs/lustre/health_check` on the Lustre Metadata Server (MDS) node and Lustre Object Storage Server (OSS) nodes to verify that `health_check` is not reporting any issues. In the absence of issues, the plugin returns *ok*; otherwise it returns *critical*. This check will catch any issues that inhibit the functioning of the previous check, *e.g.*, a request exceeds its processing timeout limit (slowness); a Lustre block device has erroneously entered a read-only state; the script encounters a Lustre source bug (LBUG).

It was important to ensure that the system's plugins could handle all possible return states appropriately. Situations arise wherein the management network goes down, effectively preventing the Nagios server from contacting a Lustre host; however, Lustre is still potentially functional on another network (*e.g.*, a separate Infiniband network). In this case, one would not want the plugin to return *critical*, but rather *unknown*, to prevent a work stoppage on a computational resource for which the true state is still unclear. There are numerous other checks and software packages to assist in determining the health of Lustre file systems, but these were not included in the Lustre-aware Moab system due to their limited utility when considering the implementation of a work stoppage, *i.e.*, not all checks that return *critical* warrant the creation of a queue reservation.

C. Nagios Event Handler

If either of these specific (2) Lustre file system checks goes into a hard-critical state, the custom event handler sends an SNMP trap to the Moab server to create a batch scheduling reservation for any computational resource that mounts the compromised file system. By only taking action while in a hard-critical state, we avoid work stoppages during the occasional false positive.

D. Moab Integration

The Lustre-aware Moab system takes advantage of the *consumable generic resource* Moab feature in order to provide a targeted work stoppage. In practice, each individual filesystem is given a consumable generic resource. Each filesystem can then be individually targeted for work stoppage based upon the specific needs of each job. Jobs that are not dependent upon a currently-problematic filesystem can continue running without interruption. Similarly, new jobs that that are not dependent upon a currently-problematic filesystem can begin execution as usual. This has obvious advantages over interrupting all jobs and preventing all new jobs from starting.

Another key part of the system takes advantage of the *submit filter* Moab feature which provides the ability to modify job attributes before jobs begin. Each job is modified at submit time to require the appropriate consumable generic resource(s) based upon the filesystem(s) mounted

on the requested compute resource. In other words, if a job requested *cluster A*, and *cluster A* only has *filesystem B* mounted, only the consumable generic resource *filesystem B* would be added to the job requirements via the submit filter. Moab does provide the ability to request multiple consumable generic resources to accommodate multiple filesystems being mounted on a particular compute resource.

SNMP traps are the mechanism by which the Nagios server communicates filesystem issues to Moab. The Moab server was equipped with a locally-developed custom SNMP MIB and trap configuration which contained (2) new events `createReservation` and `removeReservation`. Using the particular filesystem data passed via the SNMP trap, the requested event calls a script which takes appropriate action on the Moab server to create or remove a reservation against the consumable generic resource. Optionally, on a `createReservation` event, the script can requeue currently running jobs on any compute resources containing the consumable generic resource. While this option is not currently being used in production, the other features are; new jobs are being prevented from starting until filesystem issues are cleared and the Nagios server sends a `removeReservation` event.

V. Conclusion

The automated workflow software employed by Gaea end-users, combined with the public scrutiny to which climate research data are sometimes subjected, create a high premium on data integrity which mandates a unique support environment to protect NCRC research outcomes. ORNL and NOAA staff have created and implemented a series of useful tools as a response to these needs that are generally useful even outside of workflow-oriented HPC environments. As the need for more sophisticated workflow components grows into the foreseeable future, the support solutions required will need to evolve as well.

References

[1] F. Indiviglio and D. Maxwell. The NCRC Grid Scheduling Environment. *Cray User Group*, 2011.

[2] Nagios. *http://www.lustre.org*.

[3] R. Miller, et al. Monitoring Tools for Large Scale Systems. *Cray User Group*, 2010.

[4] N. P. Cardo and C. Whitney. Reaping the Benefits of Metadata. *Lustre User Group*, 2010.

[5] Nagios. *http://www.nagios.org*.