Design, Implementation, and Experiences of Third-Party Software Administration at the ORNL NCCS

Nick Jones *and* **Mark Fahey**, *National Center for Computational Sciences, Oak Ridge National Laboratory*

ABSTRACT: *At the ORNL NCCS, the structure and policy surrounding how we install third-party applications. This change is most notable for its effect on our quad-core Cray XT4 (Jaguar) computer. Of particular interest is the addition of many scripts to automate installing and testing system software, as well as the addition of automated reporting mechanisms. We will present an overview of the design and implementation, and also present our experiences to date.*

KEYWORDS: ORNL, NCCS, Cray XT4, software management

1. Introduction

ORNL NCCS

The National Center for Computational Sciences (NCCS) was established at Oak Ridge National Laboratory (ORNL) in 1992 [1]. In 2004 the Secretary of Energy designated the center as the Leadership Computing Facility for the nation, with the mission of delivering world class computing facilities for open scientific research.

The primary goal of the NCCS is to support open science and research in areas of interest to the Department of Energy (DOE) deems worthy of investigation. This is primarily accomplished through the DOE's Innovative and Novel Computational Impact on Theory and Experiment (INCITE) program, with over 145 million processing hours awarded on ORNL systems for the 2008 year [2]. INCITE is one of the United States' primary programs for computationally intensive research.

Supporting INCITE users and their corresponding software requirements is one of the primary goals of NCCS staff. Each INCITE award recipient is granted a set amount of processor hours during which they may run on NCCS systems. Since INCITE awards are made to users from research institutions all over the world, there is very little homogeneity between the types of applications that are run on NCCS systems.

NCCS staff members support multiple programming languages, compilers, and libraries on each system. For example, the NCCS currently supports over 60 different libraries, tools, and applications on our Cray XT4 system. For most applications and libraries the NCCS supports multiple versions of the software and multiple builds of each version. In addition, the NCCS currently supports more than three significantly different system architectures including Power PC (IBM Blue Gene/P), vector (Cray X1E), and x86/Opteron (Cray XT4). Furthermore, the NCCS staff will be preparing for a new petascale Cray XT5 system that will likely be installed in late 2008.

Software Management System

The authors of this paper have gained a large amount of experience managing heterogeneous software environments during their tenure at the NCCS. They knew that many of the tasks they encountered on a daily basis could be automated or streamlined if a consistent structure for software management was put in place, consistently used, and enforced. Over the course of several months a system was designed based on both personal experience and input from colleagues. (Hereafter referred to as *swtools.*) Beginning in January 2008, the swtools infrastructure began to be actively developed. The new design incorporates automation of new software installations and testing of existing software installations. Additionally, there are several interesting mechanisms in place to provide information on currently installed software to end users of the systems. In what follows, the details of the design and implementation of the software management system are given, along with a description of the challenges that were faced along the way.

2. Description of Problem and Design Goals

Previous Software Management

Prior to the deployment of swtools, the NCCS used a file system named `/apps`, an NFS cross-mounted directory. In `/apps` all libraries, tools, and applications were installed for all systems. Unfortunately, there were problems which hindered the long-term usefulness of the `/apps` directory. Specifically, the NCCS lacked a standardized workflow for software installations and lacked reporting data for already installed applications. The lack of a standardized workflow led to inconsistency in naming and directory structure, which made it hard for users to use the systems. This made tasks such as inventories and upgrades significantly more labor intensive. Since all reporting data was collected by hand, inventories and other user support documents were prone to becoming quickly out of date.

Originally, the `/apps` filesystem was designed as a method for organizing and maintaining software. A set of rules were defined that would enforce naming and version hierarchy within the tree. At the NCCS, there were a large number of staff who contributed software installations in `/apps`. Since there were so many different people using the system, small idiosyncrasies were introduced into the naming and organization of `/apps`. For example, x86 was used to label installs for machine A with interconnect Y, which would not work on another x86 machine, B with interconnect Z. Also, when the list in `/apps` got long, few noticed that there were duplicate packages trees due to capitalization (`APP1` versus `App1`). Over time, these small errors compounded and made it harder for users to find software within the tree. These errors were not noticed initially, because there were no tools in place to automate and enforce the rules that defined the `/apps` hierarchy.

In addition to these problems of organization, there was also a lack of documentation for the software that was installed. Although the installed software was still usable, upgrades to future versions often required a significant amount of effort to discover the proper compiler flags and build options. This was clearly inefficient since the work to discover this information had already been done by the installer of the previous version of the software.

Additionally, there were no automated reporting mechanisms in place for use with `/apps`. This meant that all user documentation, inventory, and other reporting data were created by hand. This led to additional work because the installers and the people writing the user documentation were often different people. This made it even harder to keep documentation and inventories up to date.

Goals for New System

After having gained significant experience with the `/apps` tree, the authors had several distinct goals in mind when they approached the task of designing the NCCS' new software management system swtools. The main goal of the new design was to have a system that was strongly hierarchical, with rules for naming, installing, and documenting that could be enforced in automated way. In addition, the authors sought to automate as much of the software maintenance process as possible. Several new features were implemented to accomplish this goal.

Foremost, the authors wished to be able to build, link, or test any installed application on any system at any time. One problem that all software installers and maintainers face is the problem of upgrades, whether those happen to be compiler upgrades, operating system upgrades or application upgrades. Any time one of those events occurs, a huge amount of work has to be done. At the very least, the upgrade has to be installed, and all applications that depend on it must be retested. In the case of OS upgrades, the maintainer may have to relink a significant number of the applications on the system in order to fix problems relating to new system libraries. Compiler upgrades are somewhat simpler in that they usually do not break any of the existing system applications, but nonetheless, any third party libraries that are provided on that system should be compiled with the new compiler – a simple regression test of the compiler.

The authors also sought to automate the collection of as much reporting data as possible. In the new system, all NCCS-provided documentation will be written by the application installers. This theoretically will keep the documentation as up to date as possible. Also, inventory and user documentation will be kept up to date dynamically. That is, the same documentation that the installer writes for local users of the system will be made available online, and all web based inventory information will be dynamically updated. This begs the question, what keeps the application installer from writing poor documentation? A workflow has been defined that includes a review process by a software administrator. This administrator will ultimately be responsible for making sure all applications and packages conform to the rules that have been set out, and that all information presented is high quality in nature.

3. Implementation

Directory Structure

When designing the directory structure for the new software management system, we chose a hierarchical structure that was strongly tied to having individual documentation and software installations for each machine (Figure 1.) At the root of the system, we created a single folder for each machine (Figure 2). A compromise was made, however, concerning the issue of folders for machines that are nearly identical in usage and hardware. The primary examples of this are the Cray XT4 systems. The NCCS is fortunate to have multiple Cray XT4 systems, and because of this we chose to create a single directory for all XT4 software and documentation. This is in an attempt to find the solution that balances the duplication of work and the risk of incompatibility across machines. Thus all of the XT4 systems share executables and software. The majority of the other machines within the NCCS do not share any software and use only software that was specifically compiled for that system.

Inside each machine directory, there is a folder for each application that is installed on the machine (Figure 3.) In addition, we have a directory called `modulefiles` that also resides at this level. It is used to store modules (for use with the GNU Modules application [3]) for each piece of installed software. At this level of the directory structure we use only generic, version unspecific software names. For example, in our Cray XT4 tree, we have folders for `python` and `szip`, and not `python2.5.1` or `szip2.0`.

Inside each application folder, there is a folder for each version of that software. Each version folder is named using the version number. In the case of more complexly-versioned applications (e.g. GAMESS which uses a combination of dates and version numbers), a version naming standard on a per-application basis is adopted and used from then on. A folder for each build of the software is created inside the version directory. These build directories are named using the os_compiler[_options] naming format. For example, a build on Jaguar might be named `sles9.2_pgi7.0.7_i8`. The first abbreviation is for the OS, in this case Suse Linux Enterprise Server 9.2. The second abbreviation is for the compiler, which in this case is the Portland Group Incorporated group of compilers. The last abbreviation in this case is i8, which signifies that this build has been compiled with eight-byte integer support.

```
Directory Structure:

  <root>/<machine>/<application>/<version>/<build>

Example:
      /sw/xt/hdf5/1.6.7/sles9.2_gnu4.2.1
```

Figure 1: Example Directory Stucture

Files used by swtools

Inside each level of the software management system there are several files that are used by the system to store information and automate tasks. In Figure 4 you can see that the first level at which these files occur is at the application level. Each application folder contains a folder for each version of the application installed on the machine. Additionally, there are four files used by the software tools. The first file, `.check4newver` stores a date. The date in the file is the date on which swtools should check for a new version of the software. By default, the system uses a time span of 90 days in between version checks. However, this can be modified to any time span that the installer finds appropriate. The `swversion` script (which will be run in cron) will traverse all of these files and email the owner if it is time to check for a new version.

```
Root Level (i.e. Files inside the root folder):
    <Machine Folder 1>
    <Machine Folder 2>
    <Machine Folder 3>
    …
```

Figure 2: Bottom Level File Hierarchy

```
Machine Level:
    modulefiles
    <Application Folder 1>
    <Application Folder 2>
    <Application Folder 3>
    …
```

Figure 3: Machine Level File Hierarchy

```
Application Level:
    .check4newver
    description
    support
    versions
    <Version Folder 1>
    <Version Folder 2>
    …
```

Figure 4: Application Level File Hierarchy

Another file is `description`. This file is an html description of the application and its usage. It uses basic standard html tags such as `<h1>`, `<h2>`, `<p>`, and `<pre>`. No styling is incorporated into the file beyond this basic level. The idea behind this file is that the installer of the application will most likely be the person most qualified to write end user documentation about the application. The installer is most likely familiar with the package, and is certainly familiar with any idiosyncrasies particular to the builds on this system. Thus it is hoped that there will be only one source of documentation for both local system documentation and also external web-provided documentation. The only caveat to this design is that local shell users of the documentation may have to overlook some HTML syntax. However, since all styling of the documentation is handled through CSS, the `description` files are generally very readable regardless of the html embedded within them.

The third file found in every application directory is the `support` file. This file is used by swtools to document the support level that the NCCS will provide to end users for this application. There are four valid keywords that may be used in this file: `nccs`, `vendor`, `nccs+vendor`, or `unsupported`. The `nccs` keyword implies that this application was installed by the NCCS and that it is a supported application. The `vendor` keyword implies that the application was provided with the system and that the system vendor will provide support. `nccs+vendor` signifies that both a vendor-provided version and an NCCS-provided version are installed on the system. The version the user is running will determine which avenue support requests will be directed to. Lastly, the `unsupported` keyword implies that the NCCS will not provide support for this application. This is used for non-essential packages that are often installed, but has peripheral usage. Examples of these might be IDEs or rarely-used (at the NCCS) applications. In these situations, the NCCS staff will make a good faith effort to install the application correctly and insure it is working. However, if problems are encountered, only minimal support will be given.

The fourth file in the application directory is the `versions` file. The `versions` file is used by swtools to determine the current, development, and deprecated versions of each application. This file contains version tags, identifying words combined with version numbers that correlates to those words. All applications posess the `current` tag. Thus an application might have a line that says `current: 4.3b` in its versions file. It is extremely important that it is always up to date, because it is used by swtools to perform actions on all applications a specific tag. Thus one common action might be to retest all applications with the `current` tag.

At the version level there are no special files used by swtools. There is simply a folder for each build present within the version folder (Figure 5.)

```
Version Level:
    <Build Folder 1>
    <Build Folder 2>
    <Build Folder 3>
    …
```

Figure 5: Version Level File Hierarchy

At the build level there are seven separate files that are used by swtools (Figure 6.) The first three files [`rebuild`, `relink`, and `retest`] all serve a similar purpose. They are scripts written by the application installer to perform the eponymous action. For example, a rebuild script should completely clean out any existing installations and then build the application anew. One key component of this design is the use of the GNU Modules application in order to set up programming environments. By using modules, a script that was written for an older version of a compiler can easily be migrated to use a newer programming environment The only thing that is usually necessary is to modify which programming environment module is loaded.

The next file found at the build level is the `.owners` file, which is used by the system to document which staff member is currently responsible for maintaining this package. This file is automatically created when the application is installed, with the installer's username set as the initial value.

The fifth file used by swtools at the build level is the `status` file. This is a vital piece of the system; it lets NCCS staff know if an application is ready to be used and in complete working order. The `status` file is created by the `retest` script. Based upon whether the application passes its own internal tests we set the status file to either `verified` or `unverified`. If the `retest` script fails before the tests complete, then no status file will be created.

`build-notes` is the sixth file used by swtools at the build level. build-notes is not a mandatory file; it is a place for installers to document any special steps they had to take when compiling this application. If no special steps were taken, then the installer is free to leave this file empty.

Finally, the `dependencies` file is used by swtools to document any applications that are needed in order for this application to be built. Currently, this information is stored for reference use. An area of future work is to implement a graph algorithm to fully resolve all dependencies on the entire software tree.

```
Build Level:
    rebuild
    relink
    retest
    .owners
    status
    build-notes
    dependencies
    <Source Folder>
```

Figure 6: Build Level File Hierarchy

Exceptions

There is one additional file that plays a key role in the swtools system. The `.exceptions` file can be placed inside either an application or version folder. There are three keywords that can be used in conjunction with the `.exceptions` file: `controlled`, `noweb`, and `vendor`.

Controlled implies that this application or version of the software is not open to the general user base of the system. In these situations, access will usually be controlled via group permissions. One caveat for structuring the system in this way is that only people in the group that owns the directory will be able to rebuild, relink, or retest the application. In some situations, not all

NCCS staff will have the proper permissions to run these actions. Thus when we are doing large batch style operations, unless the person running the batch job is a member of all controlled groups, they will be unable to perform actions on some files.

The `noweb` keyword implies that this application or version should not appear on the website. This exception is used by `swreport` when generating html documentation.

The third exception that we define is the `vendor` exception. This vendor exception is used by the swtools so that versions or applications that are marked as vendor provided appear only on the web site and are ignored by all other aspects of the swtools infrastructure. For example we often use this keyword in conjunction with vendor provided compilers such as pathscale and gcc. When using this keyword, we still write a `description` file for end user documentation, and that file still appears online.

4. Command Line Utilities

After having defined goals that needed to be accomplished, the authors were then faced with the challenge of developing tools that could accomplish these tasks. Python was chosen as the primary language for the system, although many shell scripts were used in the final product. This decision was based on several factors. First, python was already installed and being used by other projects at the NCCS. Secondly, python offers a very clean syntax and is easily tested. Additionally, python offers an extensive library of modules to perform shell and OS level operations. However, Python 2.5 is required for some key functionality not available in earlier versions.

swaddpackage and swaddbuild

`swaddpackage` and `swaddbuild` are two scripts that were developed to ease the process of installing new applications. The first script, `swaddpackage` is used by staff members to install a new application. The script creates the application directory, automatically placing it within the proper architecture directory. To determine the correct architecture, it automatically detects the machine upon which the script is running. After creating the application directory, `swaddpackage` copies several template files into the newly created directory and populates them with as much information as possible.

The second script, `swaddbuild` performs a similar function for builds. `swaddbuild` creates the appropriate template files in the build directory.

aIn addition to this, one important role that `swaddpackage` and `swaddbuild` fulfill is correctly setting permissions. On NCCS systems, by default most files are not created as group writeable. However, in order to accommodate the large number of staff who need to be able to modify installed software, all files managed by swtools need staff group write permissions.

swbuild, swlink, and swtest

`swbuild`, `swlink`, and `swtest` all perform very similar functions. Each script executes the appropriate build script, i.e., `swbuild` executes `rebuild` scripts and `swlink` executes `relink` scripts. They support a number of advanced features that make doing large operations significantly easier. For example, each script automatically fixes all file permissions after running the corresponding [`rebuild`, `relink`, `retest`] script.

swversion

`swversion` is a tool to be used by NCCS staff to proactively update as many applications as possible. When an application is installed, a date is recorded in that applications `.check4newver` file. `swversion` is a script that is designed to run in cron. When run, it checks all applications' `.check4newver` file for a due date, that is, a date upon which we need to check for updates to this package. If the application has passed its due date, then an email will be sent to the owner (as defined in the `.owners` file).

swduplicate

`swduplicate` is a script that is designed to automate the process of performing upgrades and updates. Using `swduplicate`, you can take an already existing build and move it to another location. When the build is moved, you can then substitute the existing module commands with new module commands. Thus, if a new compiler version is installed, we could copy all builds made with Pathscale 2.5 into new directories appropriately renamed. Additionally, as they are copied, the appropriate sections of the `rebuild`, `relink`, and `retest` scripts will be modified so that they automatically load the new environment. After performing this operation, we could then execute the newly modified `rebuild` and `retest` scripts to perform a test of the new programming environment.

swreport

All reporting features of swtools are encapsulated within `swreport`. Currently, `swreport` supports three modes of operation: `conform`, `html`, and `text`. One of the above three keywords is used as an argument on the command line when `swreport` is run.

When the `conform` argument is specified, `swreport` generates a report on stdout that lists problems that it has found with currently installed builds (see Figure 7.)

When the `html` argument is specified, `swreport` generates html pages for all software installed using the swtools infrastructure. Figure 8 shows one of the generated status pages. It lists all software installed on all machines, and presents a table showing which machines each piece of software is installed upon.

Alternatively, `swreport` can generate a category-based view of the tables. This is particularly useful for users who are trying to get a general feel for what software is available on a particular system.

Example Output (abridged):

```
/xt
        /arpack (support status: nccs)
        /atlas (support status: unsupported)
        /aztec (support status: unsupported)
        /blacs (support status: vendor)
            no modulefiles folder
        /blas (support status: nccs+vendor)
        /cmake (support status: unsupported)
        /craypat (support status: vendor)
            no modulefiles folder
        /doxygen (support status: nccs)
            /1.5.4
                 /sles9.2_gnu4.2.1(owner: nai)
                      status : not gW
        /ferret (support status: unsupported)
        /fftpack (support status: unsupported)
        /fftw (support status: nccs+vendor)
            /2.1.5
              version does not have modulefile
            /3.1.1
              version does not have modulefile
```

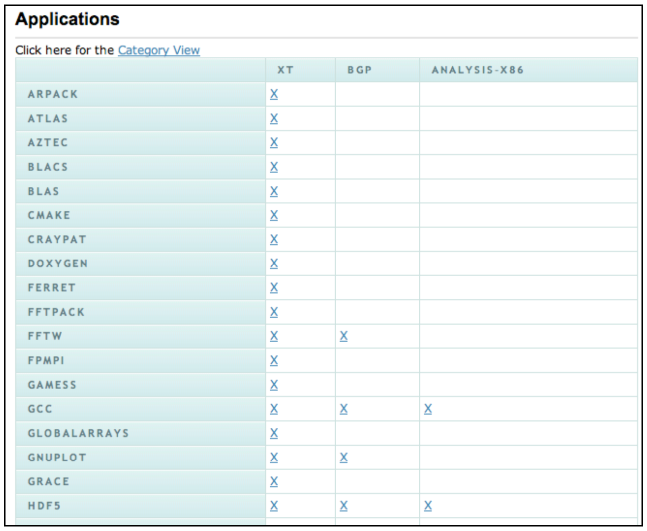
Figure 7: Example Output from `swreport conform`

Additionally, `swreport` takes the html `description` files that were written by the original installer and generates complete `description` pages for each application. In addition to presenting the information that the installer wrote, the application page shows the support status of each individual piece of installed software. At the bottom of each application status page is a list of available builds. The tables shows what versions are available, and what builds of each version are available. Next to each build the status of the build is displayed. We display a "v" for verified and a "u" for unverified. This is the build status inform the `status` file generated by the `retest` script.

`swreport text` is the third type of report that is generated by `swreport`. It generates a listing of all currently installed applications on all machines. This listing shows the status, support level, and build owner in an easily accessible format. The purpose of the `text` keyword is to generate a listing that will quickly allow NCCS staff to ascertain the status of all installed software on all machines.

Development Challenges

During the development of the swtools, a number of challenges were encountered. One of the most daunting challenges was the problem of file permissions. During the process of testing this system, it was discovered that several applications supports incorporate `chmod` commands into their `make` or `make install` scripts. This caused numerous problems – the swtools system is structured around the idea that each staff member will install applications using their own username, not using elevated root permissions. Because of this, it was imperative that all files remain group writeable at all times. To fix this, it was necessary to incorporate `chmods` into our `swbuild`, `swtest`, and `swlink` scripts.


Applications

Click here for the Category View

	XT	BGP	ANALYSIS-X86
ARPACK	X		
ATLAS	X		
AZTEC	X		
BLACS	X		
BLAS	X		
CMAKE	X		
CRAYPAT	X		
DOXYGEN	X		
FERRET	X		
FFTPACK	X		
FFTW	X	X	
FPMPI	X		
GAMESS	X		
GCC	X	X	X
GLOBALARRAYS	X		
GNUPLOT	X	X	
GRACE	X		
HDF5	X	X	X

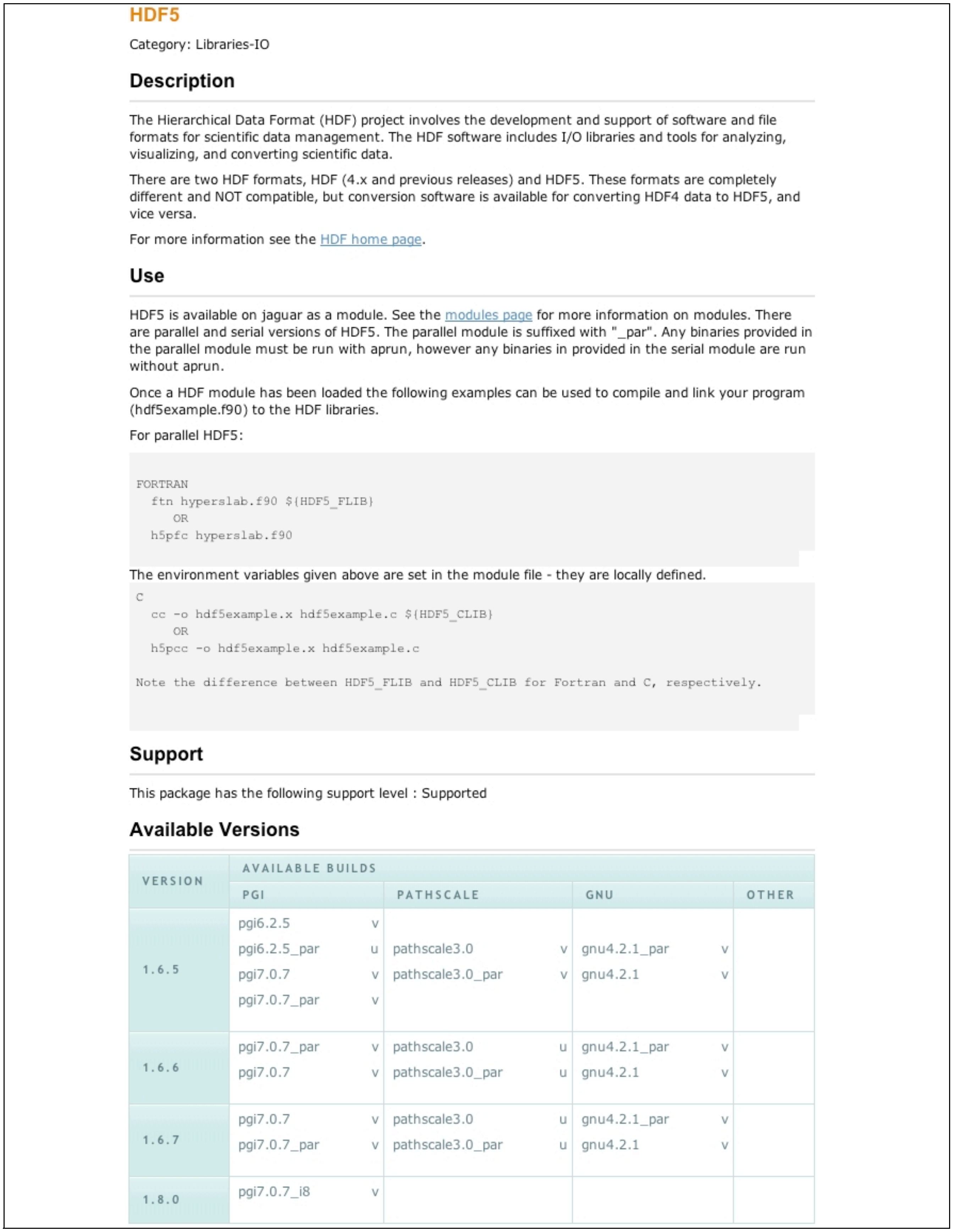
Figure 8: Screenshot of Inventory Page from www.nccs.gov

Advanced Functionality

One of the key design goals for our system is the ability to do large batch-style operations, and to combine multiple actions into a single command. To do this, a script known as `swdriver` was developed. One example of a batch operation that is routinely performed is a "duplicate build test." This action takes a set of current builds and copies them to a new location. After copying, they are then built anew, and finally they are tested. This is a great way to perform basic regression testing on all applications without destroying the current installed copies.

5. Experiences to Date

The software tools were first shown to other NCCS staff beginning on March 4, 2008. This was done in anticipation of migrating all software currently installed in the `/apps` filesystem to the new `/sw` filesystem that is managed by the swtools.



HDF5

Category: Libraries-IO

Description

The Hierarchical Data Format (HDF) project involves the development and support of software and file formats for scientific data management. The HDF software includes I/O libraries and tools for analyzing, visualizing, and converting scientific data.

There are two HDF formats, HDF (4.x and previous releases) and HDF5. These formats are completely different and NOT compatible, but conversion software is available for converting HDF4 data to HDF5, and vice versa.

For more information see the HDF home page.

Use

HDF5 is available on jaguar as a module. See the modules page for more information on modules. There are parallel and serial versions of HDF5. The parallel module is suffixed with "_par". Any binaries provided in the parallel module must be run with aprun, however any binaries in provided in the serial module are run without aprun.

Once a HDF module has been loaded the following examples can be used to compile and link your program (hdf5example.f90) to the HDF libraries.

For parallel HDF5:

```
FORTRAN
  ftn hyperslab.f90 ${HDF5_FLIB}
     OR
  h5pfc hyperslab.f90
```

The environment variables given above are set in the module file - they are locally defined.

```
C
  cc -o hdf5example.x hdf5example.c ${HDF5_CLIB}
     OR
  h5pcc -o hdf5example.x hdf5example.c

Note the difference between HDF5_FLIB and HDF5_CLIB for Fortran and C, respectively.
```

Support

This package has the following support level : Supported

Available Versions

VERSION	AVAILABLE BUILDS						
	PGI		PATHSCALE		GNU		OTHER
1.6.5	pgi6.2.5	v					
	pgi6.2.5_par	u	pathscale3.0	v	gnu4.2.1_par	v	
	pgi7.0.7	v	pathscale3.0_par	v	gnu4.2.1	v	
	pgi7.0.7_par	v					
1.6.6	pgi7.0.7_par	v	pathscale3.0	u	gnu4.2.1_par	v	
	pgi7.0.7	v	pathscale3.0_par	u	gnu4.2.1	v	
1.6.7	pgi7.0.7	v	pathscale3.0	u	gnu4.2.1_par	v	
	pgi7.0.7_par	v	pathscale3.0_par	u	gnu4.2.1	v	
1.8.0	pgi7.0.7_i8	v					

Figure 9: Example application description page

Unveiling

During the time period when the software tools were being developed, the NCCS leadership Cray XT4 system was experiencing downtime due to a major system upgrade. This afforded staff a good opportunity to make major system software changes. After the swtools system was initially unveiled to NCCS staff, each staff member was assigned a set of applications to reinstall in the new `/sw` tree using the tools.

Over a span of four weeks, approximately 60 applications were installed by 17 staff members. The swtools infrastructure worked fine with only a few bugs found. Every day as the process progressed, a report was generated listing the status of each application. This report was used by the swtools team to quickly address non-conformance with the hierarchy/rules (one of the design goals).

During this time, a feature was discovered in the SLES `man` command whereby man pages could not be found if there was a "+" somewhere in the pathname. So instead of building a new version of `man` and installing that on all SLES systems, it was decided that all build names would be changed from using a "+" to an "_". Starting on a Friday, all builds were renamed, and then scripts were used to rebuild all the application builds. This processed finished the following day. Some bugs were found in the `rebuild` and `retest` scripts, but overall this was a huge success in that over 60 packages and all their respective versions and builds were rebuilt in a day.

Jaguar Upgrade

A similar situation (where all packages are rebuilt) may occur when the upgraded Jaguar becomes available. At that time, the OS and MPI will have sufficiently changed enough that that most software may need to be rebuilt. First, everything will be tested. For those packages that fail, a determination of whether a relink or rebuild is needed will be determined on an application specific basis.

Compiler Naming Schemes

To date, the compiler names chosen for the XT have worked out well. Essentially, pgi, pathscale, and gnu are used to label all compilers. These names will likely be used on future Opteron clusters. However, on machines with IBM (BG/P) or Intel compilers, the compiler naming scheme is not so simple. Since the Fortran and C compilers on these systems are versioned independently of one another, the compiler naming scheme will have to be reworked to include both the Fortran and C compiler versions when naming builds. For example, a build on BG/P might look like cnk1.0_mpixlc9.0mpixlf11.1.

6. Future Goals

Although significant progress has been made to date, there are still several areas where the system could be expanded. Although the system was designed with HPC systems in mind, it is still a daunting challenge to deal with the many exceptions that are encountered with the extremely complex system architectures that are present in a non platform-homogenous HPC center like the NCCS. Sensitive applications, vendor applications, and special filesystems are just some of the challenges that were dealt with in the design of this system.

Complete Systems Migration

After completing the Jaguar migration, it serendipitously occurred that the NCCS had three other new systems to unveil immediately thereafter. The NCCS will unveil three new systems – Lens, Smoky, and Eugene; these are a visualization, development, and scientific (i.e. production) cluster, respectively. They are currently scheduled to be unveiled May 5, 2008 – after the publication of this paper.

Many of these systems are running Scientific Linux 5 and have an x86/Opteron-based architecture. Initially, it was thought that one of the systems would need a different MPI than the other two systems. Due to this, the first two systems were given a shared tree in the swtools infrastructure while the third one was not. However, after some time it was decided that the third machine would use the same MPI as the first two. This has created an interesting situation in that there are now three nearly identical machines, and two of the machines will share software while the third machine will not. This situation will be an ongoing test of the effectiveness of sharing software in an HPC environment.

Additional Features

The main feature that swtools currently lacks is the ability to resolve application dependencies when rebuilding large numbers of applications. Future work could be done to build an acyclic graph of application dependencies and then generate a properly ordered list of applications to build.

In addition to work that could be done resolving dependencies, additional reporting information could be generated. Currently, there is excellent information available about what software is currently installed on the system, but that information would ideally be integrated with information regarding the usage level of different applications. Ideally, the NCCS would like to know how many users are using each application, which projects those users are involved with, and what specific pieces of each application the users are using. This information would be valuable in allocating resources for future projects.

7. Conclusion

Overall, the NCCS software management system has been a great challenge and a great learning opportunity. This system has been structured to meet the unique requirements present at the NCCS; however, the authors hope that this information will be useful to other centers or organizations with similar problems.

Retrospective

The largest challenge that was encountered in developing and designing this system were the many exceptions to every rule that are present in HPC systems. This project began with a relatively simplistic model for managing software and was then modified as problems were encountered. The largest problem that anyone considering a system of this kind needs to consider are the ways in which they will deal with software that is not well suited to a management system of this type. A few examples from the experiences at the NCCS are binary applications, sensitive applications, vendor-provided applications, permissions, and more.

Final Remarks

Overall, this project has been a success. The primary goals that were set out at the beginning of the process have been achieved, and we continue to make improvements in the system. Ultimately, the system has proven itself already in that we have been able to rebuild more than 100 builds (the complete XT4 tree) in a single day. Hopefully this will inspire others to continue this work and to expand upon it.

Acknowledgments

The authors would like to thank our colleagues for their help and input during the deployment of this management system. In particular, we would like to thank Tom Barron who helped prototype some of the functionality.

This research was sponsored by the Mathematical, Information, and Computational Sciences Division, Office of Advanced Scientific Computing Research, U.S. Department of Energy under contract number DE-AC05-00OR22725 with UT-Battelle, LLC.

This research used the resources of the NCCS at ORNL, which is supported by the Office of Science of the U.S. Department of Energy under contract number DE-AC05-00OR22725.

The authors would also like to thank the HERE@ORNL program and the Oak Ridge Institute for Science and Education for their sponsorship of Nick Jones' internship.

About the Authors

Mark Fahey is a Computational Scientist in the National Center for Computational Sciences at Oak Ridge National Laboratory. He is a long-time CUG member and currently serves as the CUG Treasurer. He can be reached at Oak Ridge National Laboratory, Building 5600, Room C111, P.O. Box 2008 MS6008, Oak Ridge, TN 37831-6008, E-Mail: faheymr@ornl.gov.

Nick Jones is a Scientific Computing Intern in the National Center for Computational Sciences at Oak Ridge National Laboratory. In addition to working at ORNL, Nick is currently pursuing a double major in Computer Engineering and Computer Science at the University of Tennessee. Nick can be reached at Oak Ridge National Laboratory, Building 5600, Room A109, P.O. Box 2008 MS6008, Oak Ridge, TN 37831-6008, E-Mail: njones11@eecs.utk.edu

References

1. "Overview", National Center for Computational Sciences, http://www.nccs.gov/about/nccs-overview/

2. "Oak Ridge leads DOE INCITE Effort in 2008", National Center for Computational Sciences, http://www.nccs.gov/2008/01/17/oak-ridge-leads-doe-incite-effort-in-2008/

3. "Modules – Software Environment Management", Sourceforge.net, http://modules.sourceforge.net/

Appendix

Example Rebuild Script

```
#!/bin/ksh

############################## standard interface to /sw tools
# Input:
#   Environment variables
#     SW_BLDDIR    current directory (PWD) minus /autofs/na1_ stuff
#     SW_ENVFILE   file to be sourced which has alternate prog environment
#                  only to be used in special circumstances
#     SW_WORKDIR   unique work dir that local script can use
# Output
#   Return code of 0=success or 1=failure
##############################

# exit 3 is a signal to the sw infrastructure that this template has not
# been updated; please delete it when ready
exit 3

if [ -z $SW_ENVFILE ]; then
  ### Set Environment (do not remove this line only change what is in between)
  . ${MODULESHOME}/init/ksh
  module unload PrgEnv-pgi
  module unload PrgEnv-pathscale
  module unload PrgEnv-gnu
  module load PrgEnv-gnu
  module swap gcc gcc/4.2.1
  ### End Environment (do not remove this line only change what is in between)
else
  . $SW_ENVFILE
fi

############################## app specific section
#

if [ -z $SW_BLDDIR ]; then
  echo "Error: SW_BLDDIR not set!"
  exit 1
else
  cd $SW_BLDDIR
fi

PACKAGE=netcdf

# clear out old installation to prevent potential libtool chmod
# commands from failing when reinstalled by another person
rm -rf bin lib include doc share man etc libexec info

#clear out status file since re-making
rm -f status

cd netcdf-3.6.2
```

```
make distclean

#export CC=cc
#export CXX="CC -DMPICH_IGNORE_CXX_SEEK"
export CC=gcc
export CXX=g++
#export F77=ftn
#export F90=ftn
#export F9C=ftn
export F77=gfortran
export F90=gfortran
export F9C=gfortran
#export CPPFLAGS=-DpgiFortran

./configure --prefix=$SW_BLDDIR \
--disable-shared
#--disable-fortran-compiler-check
#--host=x86_64-unknown-linux-gnu
if [ $? -ne 0 ] ; then
  echo "$PACKAGE configure failed"
  exit 1
fi

make all
  if [ $? -ne 0 ] ; then
    echo "$PACKAGE make failed"
    exit 1
  fi

make install
if [ $? -ne 0 ] ; then
  echo "$PACKAGE install failed"
  exit 1
fi

cd ../

################################ if this far, return 0
exit 0
```

Example Retest Script

```
#!/bin/ksh

############################## standard interface to /sw tools
# Input:
#   Environment variables
#     SW_BLDDIR    current directory (PWD) minus /autofs/na1_ stuff
#     SW_ENVFILE   file to be sourced which has alternate prog environment
#                  only to be used in special circumstances
#     SW_WORKDIR   work dir that local script can use
# Output:
#   Return code of 0=success or 1=failure   or 2=job submitted
#
# Notes:
#   If this script is called from swtest, then swtest requires
#   SW_WORKDIR to be set.  Then swtest adds a unique path to what
#   user gave swtest (action+timestamp+build) and provides this
#   script with a uniquely valued SW_WORKDIR.  swtest will
#   automatically remove this unique workspace when retest is done.
#################################################################

# exit 3 is a signal to the sw infrastructure that this template has not
# been updated; please delete it when ready
exit 3

if [ -z $SW_ENVFILE ]; then
  ### Set Environment (do not remove this line only change what is in between)
  . ${MODULESHOME}/init/ksh
  module unload PrgEnv-pgi
  module unload PrgEnv-pathscale
  module unload PrgEnv-gnu
  module load PrgEnv-gnu
  module swap gcc gcc/4.2.1
  ### End Environment (do not remove this line only change what is in between)
else
  . $SW_ENVFILE
fi

############################## app specific section
#

if [ -z $SW_BLDDIR ]; then
  echo "Error: SW_BLDDIR not set!"
  exit 1
else
  cd $SW_BLDDIR
fi

PACKAGE=netcdf

#clear out status file since re-testing
rm -f status

cd netcdf-3.6.2
```

```
make test > $SW_BLDDIR/test.log 2>&1
  if [ $? -ne 0 ] ; then
    echo "$PACKAGE make test failed "
    exit 1
  fi

testspassed=`grep -G "All [0123456789]" ../test.log | awk '{total += $2} END {print total}'`
if [[ $testspassed -ne 49 ]]; then
  # error
  echo $testspassed tests passed!
  echo unverified > $SW_BLDDIR/status
  exit 1
else
  echo $testspassed tests passed!
  echo verified > $SW_BLDDIR/status
  exit 0
fi

cd ../

################################ if this far, return 0
exit 0
```