ARMY RESEARCH LABORATORY

The Computational Science Environment (CSE)

**by Jose C. Renteria, Richard C. Angelini, John M. Vines,
Kelly T. Kirk, and Eric R. Mark**

ARL-TR-4911 **August 2009**

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

DESTRUCTION NOTICE—For classified documents, follow the procedures in DOD 5220.22-M, National Industrial Security Program Operating Manual, Chapter 5, Section 7, or DOD 5200.1-R, Information Security Program Regulation, C6.7. For unclassified, limited documents, destroy by any method that will prevent disclosure of contents or reconstruction of the document.

Army Research Laboratory
Aberdeen Proving Ground, MD 21005-5067

ARL-TR-4911 **August 2009**

The Computational Science Environment (CSE)

Jose C. Renteria
Raytheon

Richard C. Angelini, John M. Vines, Kelly T. Kirk,
and Eric R. Mark
Computational and Information Sciences Directorate, ARL

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
August 2009	Progress	15 January 2008–31 December 2008

4. TITLE AND SUBTITLE
The Computational Science Environment (CSE)

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Jose C. Renteria,* Richard C. Angelini, John M. Vines, Kelly T. Kirk, and Eric R. Mark

5d. PROJECT NUMBER
9UE14C

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
U.S. Army Research Laboratory
ATTN: RDRL-CIH-C
Aberdeen Proving Ground, MD 21005-5067

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TR-4911

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES
*Raytheon, 939-I Beards Hill Rd., PMB#191, Aberdeen, MD 21001

14. ABSTRACT
In order to provide a common platform for utilizing and developing data analysis and assessment applications in heterogeneous high-performance computing (HPC) environments, the Classified Data Analysis and Assessment Center has developed the computational science environment (CSE). CSE provides a stable suite of data analysis and assessment tools, applications, and libraries that are common across most HPC environments and standard 64-bit Linux workstations. CSE also provides an experimental, cutting-edge suite of developer tools that can enhance the data analysis and assessment process. And, most importantly, CSE provides a common platform that enables the development of full-featured, portable HPC applications. This report provides an overview of how CSE works and highlights some of the benefits of CSE.

15. SUBJECT TERMS
HPC, ICE, CSE, computational environment, high-performance computing, ParaView

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Richard C. Angelini
Unclassified	Unclassified	Unclassified	UU	44	19b. TELEPHONE NUMBER (*Include area code*) 410-278-6266

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18

Contents

List of Figures

1. Introduction

Computational environments or frameworks are designed with the intent of simplifying processes associated with application/tool activation and software development. A good computational environment will allow users and developers to focus on research requirements instead of spending cycles on secondary (yet important) components not directly related to their research (e.g., portability and visualization). In this work, the Classified Data Analysis and Assessment Center introduces the computational science environment (CSE), a new module (modules 91, modules 96)-based release structure for the software platform once known as the interdisciplinary computing environment (ICE).

CSE consists of two components that can enhance the data analysis and assessment process:

1. Release CSE: A stable suite of data analysis and assessment tools, applications, and libraries.
2. Beta CSE: An experimental, cutting-edge suite of developer tools.

CSE includes the extensible data model and format (XDMF) (Xdmf02), a common data hub where high-performance computing (HPC) codes and tools can efficiently exchange data values and meaning. CSE also consists of many tools like Python, VTK, and ParaView that can assist with day-to-day visualization requirements. In addition, CSE serves as a common platform that enables the development of full-featured, portable HPC applications. CSE is available on most classified and unclassified Major Shared Resource Center (MSRC) HPC environments and is portable to most 64-bit Linux workstations.

This report outlines how an end user and developer can obtain and leverage CSE on a workstation and in HPC environments. This report can also be viewed as a collection of developer notes that describe the CSE general framework and outlines how to build, test, distribute, and utilize CSE tools and applications.

2. Background

The concept of a centralized build mechanism was initially implemented to automate building of ICE on all U.S. Army Research Laboratory (ARL)-MSRC Systems. While developing the ICE build system, many different software building tools were evaluated. Scons* was selected based

*Scons is an open source cross-platform software build tool distributed under the MIT license, scons.org.

on its powerful Python* scripting language and used the Blender† Scons implementation as the basis for the framework. The Scons toolkit was able to build all the external packages required for Blender on a variety of platforms, which reflected the requirements for ICE applications. In addition, certain system-level applications (such as Python) required for Scons to function were not consistent across computing platforms, so a bootstrap mechanism was developed to identify, compile, and install the required toolkits prior to installing Scons.

However, the initial ICE implementation did not provide a mechanism to easily modify the user's environment and resolve dependencies required to properly utilize the software. A run script was used in ICE to set environment variables and run the applications; however, this script did not easily accommodate multiple package versions or resolve version conflicts. Software updates were released on a regular schedule, and these updates required a full rebuild of the entire software structure to ensure that all changes were maintained across the suite of tools. Older versions of the software were accessible through additional scripts; however, because of ongoing, difficult-to-resolve software environment issues and library dependency errors, this methodology eventually proved to be difficult for both the users and software developers.

Once it was determined that the entire development and release methodology needed to be reevaluated, a primary requirement was to be able to easily and consistently control the application environment to ensure that the proper tools were available. It was determined that the easiest and most reliable solution was to take advantage of a solution known as modules.‡ From an end-user standpoint, modules provides a foolproof method for establishing a valid user environment and resolves package dependencies, both of which were identified as significant issues in the initial ICE release. Modules also allow for new or updated software packages to be inserted into the software release structure without disturbing existing applications or requiring a complete rebuild of the software tree.

3. Initial CSE Module Setup

CSE provides a flexible method for enabling applications, tools, and software across HPC environments. CSE simplifies the process associated with activating a particular application by leveraging modules (modules91, modules96). This section will outline how to obtain and/or initialize CSE modules.

*Python is an interpreted, interactive, object-oriented programming language under an Open Source Initiative-approved license, python.org.

†Blender is the free, open source three-dimensional content creation suite available for all major operating systems under the GNU General Public License, blender.org.

‡The environment modules package provides for an easy dynamic modification of a user's environment via module files. Released under the GNU General Public License, modules.sourceforge.net.

3.1 Machines With Modules

On the existing HPC systems (MJM, Humvee, JVN, Sketch, and Stryker) where modules are automatically integrated into the environment, users only need to know that the CSE modules exist and the procedure required to load the CSE main module (cseinit). To load CSE on the aforementioned HPC systems, type:

```
> module load cseinit
```

3.2 Machines Without Modules

On non-HPC systems, such as 64-bit Linux workstations, the modules software is included with the CSE distribution. The CSE distribution is available using rdist for RHEL4 and RHEL5 Linux workstations from a U.S. Army Research Laboratory server. For information on how to gain access to the CSE software, please contact the vis@arl.army.mil.

Once CSE is installed, modules must be initialized using one of the following methods:

1. Initialize CSE modules by adding the following to your .cshrc:

```
if (-e /usr/cta/CSE/modules/init/tcsh)then
source /usr/cta/CSE/modules/init/tcsh
module use /usr/cta/CSE/modules/COTS
endif
```

2. Initialize modules from the command line by executing the following commands:

```
source /usr/cta/CSE/modules/init/tcsh
module use /usr/cta/CSE/modules/COTS
```

---

4. CSE for the End User

The central idea of CSE is to simplify the process associated with establishing the proper environment to execute a particular application. The end user is not required to establish environment variables or modify library paths to get common applications to work properly. Although the underlying CSE tree structure in figure 1 seems complex, the end user will interact with CSE via a simple series of module commands.

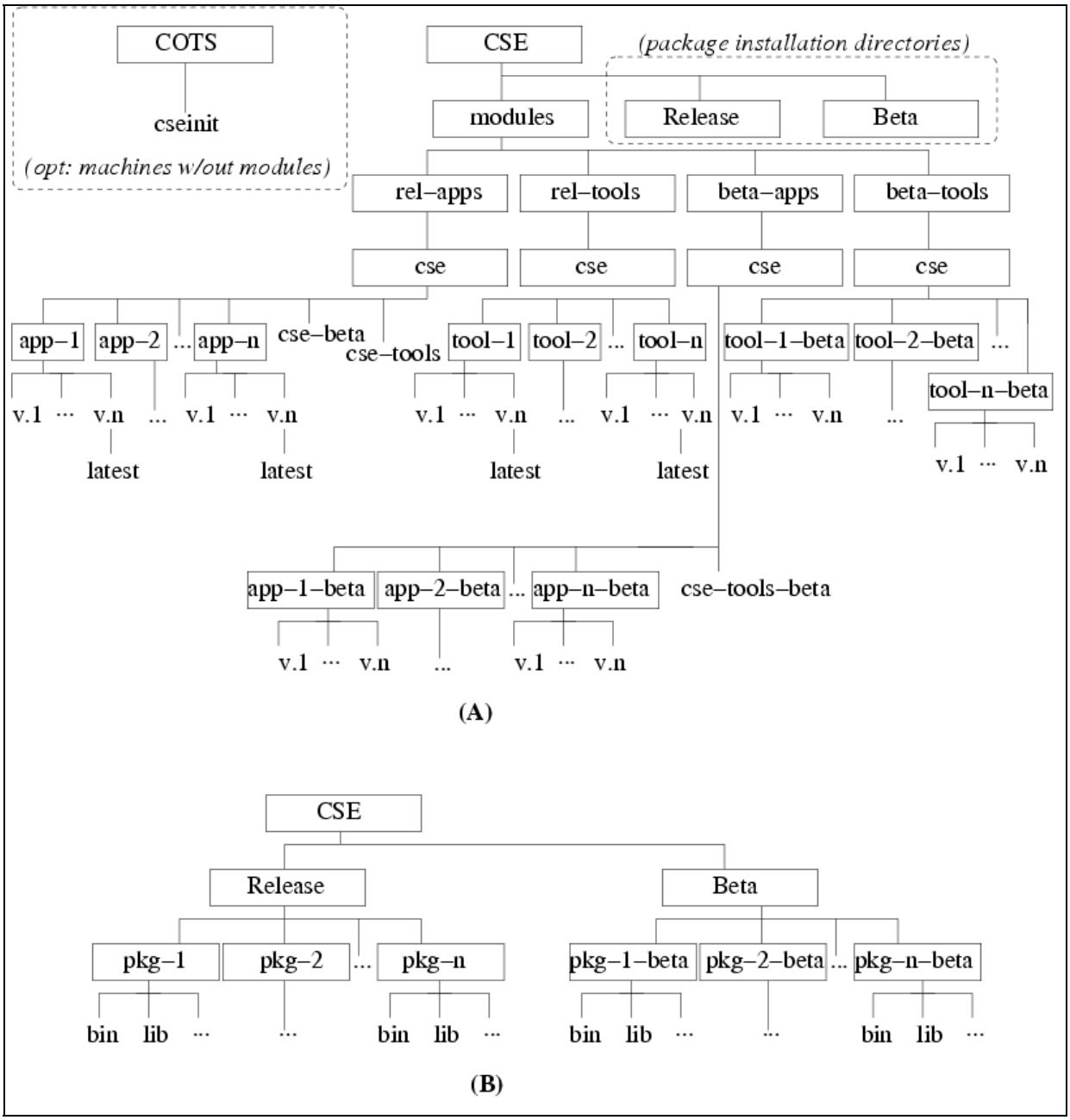

Figure 1. The CSE layout. (A) An overview of the complete CSE layout structure with an emphasis on the module layout. (B) The package layout, the location of the bin, lib, and other directories associated with each CSE package.

Modules are divided into two main components (release and beta), with two subcategories (applications and tools):

1. CSE release modules provide packages that have been formally released and tested. Each released package is considered to be stable and most likely will not change.

a. Release applications:

- paraview - visit

b. Release tools:

- cmake - mesa - qt -pryo

- tcltk - pympi - vtk -openal

- scons - hdf5 - pyqt -freealut

- libxml2 - python - sip -xsound

- svn - lyx - numpy -pyro

2. CSE beta modules provide packages that have not been formally released or tested. In most cases, these packages are the latest development version of the package. The packages are not guaranteed to be stable and will most likely change due to ongoing development updates.

a. Beta applications:

- paraview_beta

b. Beta tools:

- vtk_beta - xdmf_beta - xframework_beta

-icespy_beta -iceutil_beta - CUDA_toolkit

Note that loading multiple modules (release, beta, or both) can raise conflicts when dependencies point to different versions of the same package, e.g., qt-4.2.3 and qt-4.3.3. If a conflict arises, it is the user's responsibility to satisfy the module requirements by unloading and loading the correct sequence of modules. However, in some cases, modules cannot coexist. To illustrate how one can load different CSE release and beta packages, several examples are provided in the following sections.

4.1 End-User Examples

4.1.1 Loading a Release CSE Application Module

To view and load a CSE release application module:

1. Load CSE module:

```
> module load cseinit
```

2. List the modules:

```
> module avail
```

3. Load desired module:

```
> module load cse/paraview/latest
```

Loading the main CSE module (`cseinit`) will make the release application list (ParaView, Visit, OpenMPI ...) available to load. CSE will commonly provide several versions of the same package; by default, the latest version of a particular package is cse/package/latest. Note the `cse tools` and `cse beta` modules will be available to load.

4.1.2 Loading a Release CSE Tools Module

To view and load a CSE release tools module:

1. Load CSE module:

```
> module load cseinit
```

2. Load CSE tools module:

```
> module load cse-tools
```

3. List the modules:

```
> module avail
```

4. Load desired module:

```
> module load cse/python/latest
```

Loading the CSE tools module (`cse-tools`) will make the release tool list (`CMake, Python, TclTk`…) available to load.

4.1.3 Loading a CSE Beta Application Module

To view and load a CSE beta application module:

1. Load CSE module:

```
> module load cse
```

2. Load CSE beta applications module:

```
> module load cse-beta
```

3. List the modules:

```
> module avail
```

4. Load desired module:

```
> module load cse/paraview_beta/04.22.08
```

Loading the CSE beta module (`cse-beta`) will provide a list of CSE beta applications available to load. The beta applications are not fully tested but may provide the most current features available. Note the `cse-tools-beta` will be available to load.

4.1.4 Loading a CSE Beta Tools Module

To view and load a CSE beta tools module:

1. Load CSE module:

   ```
   > module load cseinit
   ```

2. Load CSE beta applications module:

   ```
   > module load cse-beta
   ```

3. Load CSE beta tools module:

   ```
   > module load cse-tools-beta
   ```

4. List the modules:

   ```
   > module avail
   ```

5. Load desired module:

   ```
   > module load cse/vtk_beta/04.22.08
   ```

Loading the CSE beta tools module (`cse-tools-beta`) will provide a list of CSE beta tools available to load.

4.2 End-User Benefits

CSE has several end-user advantages, including:

1. Due to the nature of modules, the end user is no longer responsible for setting environment variables or library paths necessary for proper program execution.
2. CSE applications are loaded exactly the same on all HPC platforms and workstations as illustrated in the previous examples.
3. CSE uses a larger test suite to ensure better stability of release tools and applications.
4. CSE provides a cutting-edge suite of applications and tools.

5. CSE for the Developer

Development of full-featured, portable HPC applications is considered in the CSE module philosophy by incorporating common developer principles. For example, a loaded CSE module sets the `CSE_PKGNAME_HOME` environment variable that can be used to set compiler include (`-I/CSE_PKGNAME_HOME/include`) and library (`-L/CSE_PKGNAME_HOME/lib`) flags. This allows developers to easily build or rebuild codes without having to modify the compiler flags when moving from one HPC environment to another. Therefore, an application/tool developed against CSE will compile and function on all supported CSE platforms. Developers can also build against different versions of a particular package (e.g., Python-2.4 vs. Python-2.5) via a module switch command. In the following sections, three methods of building and leveraging CSE are discussed.

5.1 Building CSE From Source

The first method is a complete CSE build. CSE has been designed to build right out of the box with minimal effort. For example, to build CSE on a x86_64 Redhat Enterprise workstation:

1. Get latest version of CSE from the source repository (see section 10.1). It is assumed that a working copy of CSE has been exported to the CSE_build directory. The name of the directory CSE_build is arbitrary.
2. Go to the CSE_build directory:

   ```
   > cd CSE_build
   ```

3. Start the CSE build:

   ```
   > ./bootstrap -a
   ```
   # (build with test)

   or

   ```
   > ./bootstrap -a -nt
   ```
   # (build without test)

After the build is complete, instructions on how to access your personal CSE will be output to the screen.

5.2 Building a CSE Package From Source

CSE is also designed to build independent packages. This feature allows developers to build only the packages that are desired and provides the platform to customize existing CSE packages. To further assist the developer, the CSE build provides feedback if dependencies are missing. For example, consider the following steps to build the pyQt CSE package on a x86_64 Redhat Enterprise workstation:

1. Get latest version of CSE from the source repository (see section 10.1).
2. Go to the CSE_build directory:

   ```
   > cd CSE_build
   ```

3. Start the CSE build:

   ```
   > ./bootstrap -a pyqt
   ```
   # (build with test)

   or

   ```
   > ./bootstrap -a -nt pyqt
   ```
   # (build without test)

In this instance, the developer forgot or is not aware of the Qt dependency on pyQt. The CSE build will terminate and provide feedback indicating that pyQt cannot build due to missing Qt.

4. Start the CSE build again:

   ```
   > ./bootstrap -a qt pyqt
   ```
   # (build with test)

   or

   ```
   > ./bootstrap -a -nt qt pyqt
   ```
   # (build without test)

In this example, two CSE packages will build. First, Qt will be configured and compiled, then pyQt will be configured and compiled. After the build is complete, instructions on how to access your personal CSE will be displayed to the screen.

5.3 Building and Integrating a New CSE Package

The third method is building a new CSE package. In this case, a developer can:

1. Obtain CSE as outlined in section 2 or build CSE from source as described in section 4.1, then create a module that interfaces a personal package with CSE.
2. Build CSE from source as described in section 2, then integrate a new CSE package.

To successfully integrate a new CSE package requires a firm understanding of the internal CSE build philosophy. Section 10, “Notes on Extending CSE,” is dedicated to this topic.

5.4 Developer Benefits

CSE offers several developer advantages, including:

1. Applications or tools developed against CSE will compile and function on all supported CSE platforms.
2. CSE modules set usable environment variables that can be used to set compiler flags.

3. Building or rebuilding code against different package versions is easier because it is not necessary to modify build definitions.
4. Custom modules can be created to interface with CSE.

---

6. CSE Principles

The CSE philosophy is derived from a set of principles that outline the build (developer), utilization (end user), testing, and distribution requirements.

6.1 CSE Developer Principles

The developer principles can be summarized as follows:

1. CSE package build and install directories must be unique. (Note: The only exception to this rule is TclTk. Tcl and Tk are built in separate directories but are installed in the same directory.)
2. Each CSE module generated by a CSE build is derived from a CSE package. This implies that a corresponding module file must be generated when a CSE package is built.
3. A CSE-generated module must set the `CSE_PACKAGE_HOME` and `CSE_PACKAGE_VERSION` environment variables.
4. To build a CSE package, all dependent modules must exist and be loaded.
5. The build and module layout described in developer principles 1 and 2 must be consistent with figure 1.
6. All modules and package directory names must use a lowercase naming scheme. Beta package modules must use an underscore, followed by the tag "beta" (`_beta`).
7. Only stable packages can be placed in the CSE release module structure.
8. All dependencies of a released CSE module must be released CSE modules or system-offered modules. Therefore, a module in the "release" area cannot have a dependency in the "beta" area.
9. The CSE release tool module's structure can only contain modules that are used by CSE release applications or critical tools (popular user tools like Python). Tool modules that do not meet this criteria must be placed in the CSE beta module structure.
10. If a CSE package is dependent on a beta CSE module, then that package must also be classified as a beta CSE module. (See rule no. 8.)

11. CSE will only support two versions of beta packages unless specifically request by a user. This implies that if a third version of a CSE beta package is available, then the oldest of these three CSE beta packages will be removed unless otherwise requested.

6.2 CSE End-User Principles

The end-user principals can be summarized as follows:

1. When a CSE module is loaded, all dependent modules must be automatically loaded.
2. A CSE module load or unload should not adversely affect standard module functionality.
3. A CSE module must detect conflicts with other CSE-loaded modules.

6.3 CSE Testing Principles

The CSE testing methodology is based on a three-phase approach as follows:

1. Test each package after initial build.
2. Test all packages after complete CSE build/install.
3. Run critical test (commonly used tools and features) frequently on installed CSE packages.

6.4 CSE Distribution Principles

1. In order to acquire local optimizations and performance enhancements (compilers, hardware-specific MPI installations, etc.), CSE will be built and released on each HPC system.
2. The 64-bit Linux workstation releases will be built on a “clean” system to ensure portability.
3. RHEL4 and RHEL5 distributions will be available for desktop workstations from an rdist master (patrick [U] and slurpee [C]). (See section 6.3 for further details.)

---

7. Notes on Building CSE

7.1 CSE Build Flags

SYNOPSIS ./bootstrap [-a] [-nt | -notest] [-i | -install |

install=1] [-f | -force | force=1][diff=1][build_mpi=1]

[build_xsound=1] [build_icespy=1] [package-name ...]

DESCRIPTION `./bootstrap` builds basic tools required to build CSE and builds CSE packages. If no flags are given, then only the basic tools required to build CSE are generated. (Note: No CSE packages are generated.)

OPTIONS The following options are recognized by `./bootstrap`:

- -nt, -notest Build CSE without running test.
- -a Build all standard CSE packages. If no package names are given, then all standard CSE package. (Note: Standard packages do not include OpenMPI, XSound, or IceSpy.)
- build_mpi=1 Build CSE version of OpenMPI. (Note: If this flag is not used, then an openmpi module must be loaded prior to running bootstrap.)
- build_mysql=1 Build a MySQL database server and a Python module ("MySQLdb") for use with MySQL.
- build_xsound=1 Build XSound. (Note: XSound adds sonification functionality to XFrameWork and is mostly intended for workstations and/or visualization systems that run demos.)
- build_icespy=1 Build IceSpy. (Note: IceSpy is not fully integrated and is only intended to build on MJM, JVN, Humvee, and Sketch.)
- diff=1 Create patch for package(s). (See section 9.2.3.)
- -f, -force, force=1 Force a rebuild of package(s).
- -i, -install, install=1 Force a reinstallation of package(s).

7.2 Building on Specific Platforms

7.2.1 Redhat Enterprise Workstation Build (x86_64)

This section provides an overview on how to build CSE on a x86_64 Redhat Enterprise Workstation. To build CSE, use the following steps:

1. Get latest version of CSE from the source repository (see section 10.1). It is assumed that a working copy of CSE has been exported to the CSE_build directory. The name of the directory CSE_build is arbitrary.
2. Go to the a CSE_build directory:

   ```
   > cd CSE_build
   ```

3. Start CSE build:

   ```
   > ./bootstrap -a build_mpi=1
   ```

7.2.2 MJM and JVN Build (x86_64-SuSE-ib & x86_64-SuSE-ib)

This section provides an overview on how to build CSE on MJM and JVN remotely from a Linux workstation. The following step will reference MJM; however, JVN can be substituted in the following build steps:

1. On Linux, open HPC ticket:

   ```
   > kinit username@ARL.HPC.MIL
   ```

2. ssh to a MJM login node:

   ```
   > ssh -Y mjm-l7
   ```

(Note: SciVis members should build in /mnt/scivis, available only on login node mjm-l7 or jvn]-l7.)

3. On mjm-l7, get latest version of CSE from the source repository (see section 10.1). It is assumed that a working copy of CSE has been exported to the CSE_build directory. The name of the directory CSE_build is arbitrary.

4. On mjm-l7, go to the a CSE_build directory:

   ```
   > cd CSE_build
   ```

5. On mjm-l7, load the gcc and openmpi modules:

   ```
   > module load ti06/gcc4.2 ti06/openmpi-1.2
   ```

6. On mjm-l7, start CSE build:

   ```
   > ./bootstrap -a
   ```

7.2.3 Humvee and Sketch Build (x86_64-SuSE-ib & x86_64-RHEL-ib)

This section provides an overview on how to build CSE on Humvee and Sketch. The following steps will reference Humvee; however, Sketch can be substituted in the following build steps:

1. On an unclassified machine, get the latest version of CSE from the source repository (see section 10.1). It is assumed that a working copy of CSE has been exported to the CSE_build directory. The name of the directory CSE_build is arbitrary.

2. Burn the CSE_build directory to a DVD. (Note: Follow proper procedures for introducing unclassified media in a classified environment. These procedures are defined in the enclave Operating Procedures document.)

3. Login to Humvee.

4. Copy the CSE_build directory from the DVD to Humvee.

5. On Humvee, go to the a CSE_build directory:

```
> cd CSE_build
```

6. On Humvee, load the gcc and openmpi modules:

```
> module load ti06/gcc4.2 ti06/openmpi-1.2
```

7. On Humvee, start CSE build:

```
> ./bootstrap -a
```

Note that building CSE on Sketch requires some additional configuration as we cannot update classified changes to the CSE repository. To build on Sketch, you may need to:

1. Modify bootstrap to recognize modules.
2. Modify patch for QT-4.2.3.
3. Modify conf file for hdf5-1.6.5.
4. Modify CMakeCache files for VTK, ParaView, and Xdmf.

Also note the ParaView-3.2.1 client on Sketch occasionally has issues terminating the openmpi associated with pvserver. In these cases, it is required to kill (using bkill command, see section 10.3) the load-sharing facility (LSF) job corresponding to pvserver.

7.3 CSE Distribution

Once CSE has been built and tested, the CSE build needs to be bundled in preparation for distribution. The update software can be distributed via a tar file or via the command "rsync." See the following two methods to bundle the CSE build.

7.3.1 Tarball Method

1. Go to the CSE root source directory CSE_build (directory where build was initiated):

```
> cd CSE_build
```

2. Go to the CSE architecture directory:

```
> cd build/ARCH
```

(Note: ARCH is the architecture that corresponds to the machine used to build CSE [e.g., x86_64-SuSE]).

3. Tar and gzip the CSE directory:

```
> tar -czf CSE.tar CSE/
```

4. Provide tar ball to system administrator or CSE installation manager for distribution.

7.3.2 Rsync Method

1. Go to the CSE root source directory CSE_build (directory where build was initiated):

   ```
   > cd CSE_build
   ```

2. Go to the CSE architecture directory:

   ```
   > cd build/ARCH
   ```

(Note: ARCH is the architecture that corresponds to the machine used to build CSE [e.g., x86_64-SuSE]).

3. Rsync the CSE directory to the installation area:

   ```
   > rsync -av CSE /usr/cta (RHEL4 installation)
   ```

   or

   ```
   > rsync -av CSE /rhel5/usr/cta (RHEL5 installation)
   ```

7.4 Building ParaView From CVS Repository for Beta Distribution

7.4.1 Updating and Preparing Required Packages for Upload to the Local SVN Repository

ParaView, ParaViewData, and VTKData are all required to be updated at the same time to facilitate the build via the CSE "bootstrap" program. The tarballs that are created and uploaded to the SVN repository on patrick must have the same download date in the filename. For instance, the latest version of ParaView downloaded from Kitware's CVS repository on 4 August 2008 would have the name "paraview-08.04.08.tar.gz", and the VTKData and ParaViewData files should also have the same "08.04.08" designation in the filename.

1. Update/download ParaView from CVS repository:

   ```
   > cvs -d pserver:anoncvs@www.paraview.org:/cvsroot/ParaView3login
   ```

   (respond with an empty password)

   ```
   > cvs -d pserver:anoncvs@www.paraview.org:/cvsroot/ParaView3coParaView3
   ```

2. Update/download ParaView data from CVS repository (required for automated ParaView testing):

   ```
   > cvs -d :pserver:anoncvs@www.paraview.org/cvsroot/ParaView3coParaViewData
   ```

3. Update/download VTK test dataset:

   ```
   > cvs -d :pserver:anonymous@public.kitware.com:/cvsroot/VTKDatalogin
   ```

   (respond with password "vtk")

```
> cvs -d :pserver:anonymous@public.kitware.com/cvsroot/VTKDatacoVTKData
```

4. Bundle up new VTKData directory and push to CSE distribution area:

   ```
   > tar cvf vtkdata-MM.DD.YY.tar VTKData

   > gzip vtkdata-MM.DD.YY.tar

   > mv vtkdata-MM.DD.YY.tar.gz

     {your-work-area}/Packages/Experimental/VTKData
   ```

   Now, alert the SVN repository that you are going to be adding a new file:

   ```
   > cd {your-work-area}/Packages/Experimental/VTKData

   > svn add vtkdata-MM.DD.YY.tar.gz
   ```

   Using your favorite editor, modify the “SConscript” file to read:

   ```
   change the variable ``pkg_install_version'' to the MM.DD.YY
   of new release
   ```

   Finally, send the changes up to the SVN repository:

   ```
   > svn commit
   ```

5. Bundle up new ParaViewData directory and push to CSE distribution area:

   ```
   > tar cvf paraview-data-MM.DD.YY.tar ParaViewData

   > gzip paraview-data-MM.DD.YY.tar

   > mv paraview-data-MM.DD.YY.tar.gz

   {your-work-area}/Packages/Experimental/ParaView
   ```

   Now, alert the SVN repository that you are going to be adding a new file:

   ```
   > cd {your-work-area}/Packages/Experimental/ParaView

   > svn add paraview-data-MM.DD.YY.tar.gz
   ```

   Using your favorite editor, modify the “SConscript” file to reflect:

   ```
   change the variable ``pkg_install_version'' to the MM.DD.YY
   of new release
   ```

   (There are four different places within the “SConstript” file where this value must be reset.)

   Send the changes up to the SVN repository:

   ```
   > svn commit
   ```

6. Bundle up new ParaView directory and push to CSE distribution area:

   ```
   > tar cvf paraview-MM.DD.YY.tar ParaView3
   > gzip paraview-MM.DD.YY.tar
   > mv paraview-MM.DD.YY.tar.gz
   {your-work-area}/Packages/Experimental/ParaView
   ```

   Now, alert the SVN repository that you are going to be adding a new file:

   ```
   > cd {your-work-area}/Packages/Experimental/ParaView
   > svn add paraview-MM.DD.YY.tar.gz
   ```

   Using your favorite editor, modify the "SConscript" file to reflect:

   ```
   change the variable "pkg_install_version" to the MM.DD.YY of
   new release
   ```

   (Note: This is the same distribution directory that we used to release the ParaViewData tarball, so this SConscript file should not need to be changed again.)

   Send the changes up to the SVN repository:

   ```
   > svn commit
   ```

7.4.2 Compiling Paraview Within the CSE Environment

1. Before compiling, you must have the proper environment established:

   On MJM/JVN:

   ```
   > module purge
   > module load ti06/gcc4.2 ti06/openmpi-1.2
   ```

   On Aspen Cluster:

   ```
   > module purge
   > module load openmpi/openmpi-gcc
   ```

   On Linux workstation:

   ```
   > module purge
   > module load cseinit cse-tools cse/openmpi/latest
   ```

2. Change directory to your the work area where you build (or intend to build) CSE and download/update CSE if you have not already done so:

```
> cd {your-work-area}
> svn up
```

3. Install the updated VTKData-beta directory:

```
> ./bootstrap -a -nt VTKData-beta
```

4. If you have previously built ParaView in this work area, you need to clean up some directories in order to ensure a proper build:

```
> cd {your-build-area}/build
> rm -rf Paraview_beta-*
```

5. If you need to recompile for the same beta release, then you will also need to clean up this directory:

```
> cd {your-build-area}/build/x86-64-RHEL/CSE/Beta
```

(X86-64-RHEL will be replaced with whichever architecture your system happens to be.)

```
> rm -rf paraview*-MM.DD.YY
```

6. If you want to rebuild the same beta release and you need to go back to the original tarball, then this directory also needs to be removed:

```
> cd {your-build-area}/build/src
> rm -rf ParaView_beta-MM.DD.YY
```

7. Build the beta version of ParaView using the bootstrap program:

```
> cd {your-work-area}
> ./bootstrap -a -nt ParaView-beta
```

---

8. Notes on Testing CSE

8.1 General Testing Information for Applications/Tools

8.1.1 pyMPI: MJM, JVN, Humvee, and Sketch

This section provides a step-by-step procedure outlining how to test pyMPI on MJM, JVN, Humvee, and Sketch while working from a Redhat Linux army.mil machine at Aberdeen Proving Ground (APG), MD. The following steps will reference MJM; however, JVN, Humvee, or Sketch can be substituted:

1. Login to HPC machine.
2. ssh to a MJM login node:

   ```
   > ssh -Y mjm-l1
   ```

3. On mjm-l1, request interactive compute nodes:

   ```
   > bsub -Is -n 2 -x -a openmpi -m mjm -W 00:30 -q interactive
   tcsh
   ```

4. On MJM compute node, load the CSE pympi module:

   ```
   > module load cse/pympi/latest
   ```

5. On MJM compute node, start pympi run:

   ```
   > mpirun.lsf pyMPI
   ```

8.1.2 VTK: MJM, JVN, Humvee, and Sketch

This section provides a step-by-step procedure outlining how to test VTK on MJM, JVN, Humvee, and Sketch while working from a Redhat Linux army.mil machine at APG. The following steps will reference MJM; however, JVN, Humvee, or Sketch can be substituted:

1. Login to HPC machine.
2. ssh to a MJM login node:

   ```
   > ssh -Y mjm-l1
   ```

3. On mjm-l1, request interactive compute nodes:

   ```
   > bsub -Is -n 2 -x -a openmpi -m mjm -W 00:30 -q interactive
   tcsh
   ```

4. On MJM compute node, load the CSE vtk module:

   ```
   > module load cse/vtk/latest
   ```

5. On MJM compute node, start VTK run.

8.1.3 ParaView: MJM, JVN, and Humvee

This section provides a step-by-step procedure outlining how to test ParaView on MJM, JVN, and Humvee while working from a Redhat Linux army.mil machine at APG. The following steps will reference MJM; however, JVN or HUMVEE can be substituted:

1. On a 64-bit Linux workstation, run the ParaView client:

   a. Load the CSE paraview module:

```
> module load cse/paraview/latest
```

b. Start ParaView:

```
>paraview -s=manual
```

2. Login to HPC machine.
3. ssh to a MJM login node:

```
> ssh -Y mjm-l1
```

4. On mjm-l1, request interactive compute nodes:

```
> bsub -Is -n 2 -x -a openmpi -m mjm -W 00:30 -q interactive
tcsh
```

5. On MJM compute node, load the CSE ParaView module:

```
> module load cse/paraview/latest
```

6. On MJM compute node, start ParaView server:

```
> mpirun.lsf pvserver --use-offscreen-rendering -rc
-ch=<clienthostname>
```

7. Now conduct the test, e.g., Compositing, CutFilter, ProbeFilter ...

8.1.4 ParaView: Sketch

This section provides a step-by-step procedure outlining how to test ParaView on Sketch while working from a Redhat Linux army.mil machine at APG. To run ParaView on Sketch:

1. On a 64-bit Linux workstation (e.g., squirt4), run the ParaView client:

a. Load the CSE ParaView module:

```
> module load cse/paraview/latest
```

b. Start ParaView:

```
>paraview -s=manual
```

2. Login to HPC machine.
3. ssh to a Sketch login node:

```
> ssh -Y sketch-l1
```

4. On sketch-l1, request interactive compute nodes:

```
> bsub -n 2 -Ip -x -W 0:30 -a openmpi -R ``span[ptile=1]'' tcsh
```

5. Set DISPLAY environment variable to 0.0:

```
> setenv DISPLAY :0.0
```

6. On Sketch compute node, load the CSE ParaView module:

```
> module load cse/paraview/latest
```

7. On Sketch compute node, start ParaView server:

```
> mpirun.lsf pvserver --use-offscreen-rendering -rc
-ch=<clienthostname>
```

8. Now conduct the test, e.g., Compositing, CutFilter, ProbeFilter ...

Note the ParaView-3.2.1 client on Sketch occasionally has issues terminating the openmpi associated with pvserver. In these cases, it is required to kill (using bkill command, see section 10.3) the LSF job corresponding to pvserver.

8.2 General Testing Errors

8.2.1 TK Testing Error and Workaround

It has been found that TK tends to produces more testing errors when using KDE, and in some instances, the test suite freezes when reaching the TK select test. These issues have not been seen when using Gnome.

8.2.2 VTK Testing Error and Workaround

VTK test hangs on test 791 Testing DistributedData-image. One will need to kill test 791 by:

1. Hit q-key, then the return-key in the terminal running the test.
2. Run top command in a different shell:

```
> top
```

3. While top is running, hit the k-key.
4. Then enter the PID associated with mpirun and hit return-key.
5. Then enter 9 and hit return-key.

8.2.3 ParaView Testing Error and Workaround

ParaView test hangs on test 84 Testing TestMetaIO. One will need to y kill test 84 by:

1. Run top command in a different shell:

```
> top
```

2. While top is running, hit the k-key.
3. Then enter the PID associated with IOCxxTests and hit return-key.
4. Then enter 9 and hit return-key.

Also note the ParaView-3.2.1 client on Sketch occasionally has issues terminating the openmpi associated with pvserver. In these cases, it is required to kill (using bkill command, see section 10.3) the bsub call corresponding to pvserver.

---

9. Notes on Distributing CSE

In order to acquire local optimizations and performance enhancements (compilers, hardware-specific MPI installations, etc.), CSE is built and released on each HPC system. Desktop distributions that support 64-bit RedHat4 and RedHat5 systems are also provided. These distributions are available via rdist from patrick.arl.army.mil. The RH4 distribution is installed in patrick:/usr/cta/CSE, and the RH5 installation is installed in patrick:/rhel5/usr/cta/CSE.

9.1 Installing Software on the rdist Master System (patrick)

Any software being pushed to patrick (using the methods defined in section 6.3) needs to be installed in the proper location. Software being pushed from the RH5 build master should be installed in the /rhel5/usr/cta/CSE tree, and software built on the RH4 master should go into /usr/cta/CSE on patrick. Any module files automatically created during the CSE build process should not need to be modified in the RH5 environment. Once the software is rdist'd to RH5 workstation, the /rhel5/usr/cta/CSE directory will appear as /usr/cta/CSE to the end user; therefore, no further changes are required.

9.2 Rdist from patrick

Setting up rdist on patrick is a two-step process. Prior to attempting an rdist, a system administrator on patrick.arl.army.mil needs to allow the desktop workstation to rdist the package(s) by modifying the proper files. The hostnames need to be added to the following files on RH4 client workstation:

`patrick:~rhel4/distfile` and `~rhel4/.rhosts`

And on RH5 client workstation:

`patrick:~rhel5/distfile` and `~rhel5/.rhosts`

9.2.1 Release Distribution Only

On the client workstation, the system administrator issues the following command (as “root”) to perform the rdist:

RHEL4 SYSTEM> `rdist.arl -C rhel4@patrick.arl.army.mil cse`

or

RHEL5 SYSTEM> `rdist.arl -C rhel5@patrick.arl.army.mil cse`

9.2.2 Release and Beta Distribution

On the client workstation, the system administrator issues the following command (as “root”) to perform the rdist:

RHEL4 SYSTEM> `rdist.arl -C rhel4@patrick.arl.army.mil cse-all`

or

RHEL5 SYSTEM> `rdist.arl -C rhel5@patrick.arl.army.mil cse-all`

9.2.3 Rdist Packages Currently Available

- cse: all packages in “release” directory and all module files.
- cse-all: all packages in “release” and “beta” directories and all module files.
- visit: visit release and associated module files.
- paraview: ParaView production release and associated module files.
- paraview_beta: ParaView from “release” & “beta” areas and associated module files.

---

10. Notes on Extending CSE

10.1 Integrating a Personal Package to CSE

To integrate a personal CSE package:

1. Obtain a copy of CSE as outlined in section 2 or build CSE from source as described in section 4.1.
2. Create a module file (myModule) that interfaces the personal package with CSE (see section 9.2.1). Note that myModule is an arbitrary name.
3. Set the module file path myModulePath (path where myModule lives):

   ```
   > module use myModulePath
   ```

4. Load your module:

```
> module load myModule
```

10.2 Creating a CSE Module File

To illustrate how to create a CSE module file, consider the following file for the CSE release module SIP, with key components for creating a release CSE-compliant module file:

- Lines 0–16: required in every module file.
- Line 17: replace SIP with the name of personal package.
- Line 18: replace 4.7.4 with the version of personal package.
- Line 19: replace $CSE_HOME/Release/$name-$version with the path to your personal package.
- Lines 20–66: required in every module file.
- Line 67: add all conflicts.
- Line 69: add commands specific to the personal package.
- Lines 70–74: required in every module file.

Note that a module file may vary from package to package and is slightly different for beta packages.

10.3 Integration Example—IceSpy

IceSpy is an example of locally developed software that was not initially developed within CSE but has been fully integrated into the environment, making it easier for the developers to maintain and extend, and greatly simplifies initialization and execution for the end user.

10.3.1 IceSpy Description

IceSpy is software that integrates the Python scripting language and the Xdmf data model with the widely used shock physics code CTH. This unique collection of components allows the end user to access the data calculated by CTH for analysis and visualization, without being hindered by the limited set of tools provided with the CTH package. Several Python-based scripts are provided to extract and convert data calculated in CTH to Xdmf, enabling the end user to analyze the results in a number of commercial and open source packages.

10.3.2 Pre-CSE Development and Execution Challenges

IceSpy is an integration of Python, Xdmf, and CTH. This combination presents challenges for both the developer and end user. IceSpy was developed under ICE, a predecessor of CSE. Environment configuration in ICE was handled by a complex setup script that attempted to

provide a common initialization source for multiple computer platforms. This in itself was difficult to maintain and, as a result, was a source of frustration for the developer and end user. These configuration issues were compounded by the fact that CTH and the rest of ICE were not built with the same compilers. CTH is built with the Portland Group C and Fortran compilers, with their associated version of MPI, and ICE was built using gcc linked against its version of MPI.

10.3.3 Using the CSE Build Process for IceSpy

To integrate the IceSpy build into CSE, the IceSpy source was copied into the CSE directory structure in /CSE/Packages/Experimental/IceSpy/81. The directory name /81 corresponds with the CTH version that IceSpy will be linked to. As described in section 10.2, the /81 directory must have a SConscript file that will be called by the main CSE SConstruct at build time. The significant elements of the following script are found in lines 25–29, where the necessary CTH environment variables are set for the creating the cth_beta module, and lines 45–47, where the required modules for building are loaded. Note that this contains the just-created cth_beta module. The ability to create and load modules at build time saves a significant amount of effort, eliminating the need for the developer to continually set paths individually prior to the build process. When the build and install processes are complete, the end user will use these modules to set up the appropriate environment for running the package.

```
1 #!/usr/bin/env python
2 import os
3 import sys
4 from Tools.scons.ice import *
5
6 Import('env')
7 myenv = env.Copy()
8
9
10
11 # configure, create and install cth module
12 #-----------------------------------------------------------------------
13 pkg_install_name = 'cth_beta'
14 pkg_install_version = '8.1'
15 install_name = pkg_install_name + '-' + pkg_install_version
16 pkg_name = pkg_install_name + '-' + pkg_install_version
17 pkg_tar = pkg_name + '.tgz'
18 release = 0
19 latest = 0
20
21 myenv.Replace( BUILDBIN = os.path.join( env[ 'BUILDBETABIN' ],
install_name.lower() ) )
22 pkg = ice_buildext.Package( pkg_name, pkg_tar, myenv )
23
24
25 extras =  [ 'setenv           CTHPATH /usr/cta/unsupported/CTH/CTH_8.1']
26 extras += [ 'setenv           CTHBIN /usr/cta/unsupported/CTH/CTH_8.1/bin']
27 extras += [ 'setenv           CTH_HOME /usr/cta/unsupported/CTH/CTH_8.1']
28 extras += [ 'prepend-path     PATH /usr/cta/unsupported/CTH/CTH_8.1/bin']
29
```

```
30 pkg.ExternalModuleInstall( pkg_install_name.lower(), pkg_install_version, env[
'MODBETATOOLDIR' ],extr        as, release, latest )
31
32
33
34 # configure, create and install IceSpy
35 #---------------------------------------------------------------------------
36
37 pkg_install_name = 'IceSpy_beta'
38 pkg_install_version = '81'
39 install_name = pkg_install_name + '-' + pkg_install_version
40 pkg_name = pkg_install_name + '-' + pkg_install_version
41 pkg_tar = pkg_name + '.tgz'
42 release = 0
43 latest = 0
44
45 modules = [ ('gcc','sys'),  ('python','required'), ('openmpi', 'required'),
46          ('vtk_beta/05.27.08','required'),('xdmf_beta/02.10.09','required'),
47             ('cth_beta/8.1','required')]
48
49 myenv.Replace( BUILDBIN = os.path.join( env[ 'BUILDBETABIN' ], install_name.lower()
) )
50 pkg = ice_buildext.Package( pkg_name, pkg_tar, myenv )
51
52
53 if not os.path.exists( myenv[ 'BUILDBIN' ] ) or myenv[ 'install' ]:
54         depend_mods = pkg.LoadModules( modules )
55         pkg.Tar(pkg_name)
56         pkg.UnTar()
57         pkg.RemoveFile(pkg_tar, './')
58
59         # configure and build
60         cmd = 'cd ' + pkg.pkg_src_dir + '/libsrc; scons install'
61         cmd += ' pgi=/opt/compiler/pgi/linux86-64/8.0'
62         cmd += ' cth='+myenv['ENV']['CTHPATH']
63         cmd += ' mpi='+myenv['ENV']['MPI_HOME']
64         cmd += ' pythonroot=' + myenv['ENV'][ 'CSE_PYTHON_HOME']
65         cmd += ' xdmfroot='   + myenv['ENV'][ 'CSE_XDMF_BETA_HOME']
66         cmd += ' vtkroot='    + myenv['ENV'][ 'CSE_VTK_BETA_HOME']
67         cmd += ' hdf5root='   + myenv['ENV'][ 'CSE_HDF5_HOME']
68         cmd += ' release=' + pkg.prefix
69         pkg.Run( cmd )
70
71         # configure and install
72         version  = sys.version_info
73         PythonVersion = "%d.%d" % (version[0], version[1])
74
75         # change group to cth
76         cmd = 'chgrp -R cth ' +   pkg.prefix
77         pkg.Run( cmd )
78
79         extras = []
80         extras += depend_mods
81         extras += ['prepend-path  PYTHONPATH  $prefix/lib/python'+PythonVersion ]
82         extras += ['prepend-path  PYTHONPATH
$prefix/lib/python'+PythonVersion+'/lib-dynload' ]
83         pkg.ModuleInstall( pkg_install_name.lower(), pkg_install_version, env[
'MODBETATOOLDIR' ],
84                            extras, release, latest )
85
```

```
86         if myenv[ 'test' ] == 'no':
87                 myenv[ 'test' ] = 'yes'
88
89 if myenv[ 'test' ] == 'yes':
90         pkg.MakeTest( pkg_name )
91
```

10.3.4 Adding IceSpy to the Main SConstruct

The code used to add IceSpy to the main SConstruct file at the base of the build file structure is slightly different than what is described in section 10.2.2.2. Because IceSpy is still a work in progress and its integration into CSE is not complete, it is listed in the main SConstruct as an option. Building IceSpy requires the developer to specifically request it on the bootstrap command line using "build_icespy=1."

The code added to the SConstruct is in the section for Options beginning with the following:

```
opts = Options( "ext_pkg.conf" )
```

This line adds IceSpy to the list of potential options available for building:

```
opts.Add( BoolOption( "build_icespy", "Build a local copy of icespy", 0 ) )
```

Adding the following code after defining the list of Release and Experimental Packages will evaluate the Boolean and add IceSpy to the list of Experimental packages, if requested.

```
if env[ 'build_icespy' ]:
    Experimental_Packages += [( 'IceSpy-beta',
'Packages/Experimental/IceSpy/81/SConscript' ) ]
```

Finally, the actual build command will look like the following:

```
./bootstrap –a build_icespy=1
```

10.3.5 Running and Testing IceSpy

When the build process has been successfully completed, the developer or end user is able to access IceSpy by loading the appropriate modules. This example uses a build directory structure that is not part of the main CSE build typically found in /usr/cta/CSE, hence the `module use` command; however, the modules to be loaded would be very similar.

```
module use /usr/people/username/CSE/02.10.09/build/COTS  (points to a local build of
CSE)
module load cseinit
module load cse-build
module load cse-beta
module load cse-tools-beta
module load cse/icespy_beta/81
```

Upon final implementation, loading the *cse/icespy_beta/81* module file will load the correct compilers, MPI version, CTH version, and Python paths and eliminate the usual requirement to troubleshoot conflicts and path errors, reducing end-user frustration and the need for developer intervention.

10.4 Integrating a New CSE Package

CSE can be extended by integrating new release or experimental packages. To add a new CSE package, one needs to consider three points:

1. Create directory structure for custom package, then store source and configuration files in the new directory.
2. Configuration files for scons:
    a. Generation of an SConscript file.
    b. Update main SConstruct to point to new SConscript file.
3. Write patch if needed.

These points are discussed in detail in the next sections and should be referenced when integrating a new CSE package.

10.4.1 Create CSE Source and Layout Structure

The directory structure that defines how new packages should be integrated is illustrated in figure 2. The CSE source directory is the same for the release or experimental packages.

The CSE source directory has two valid tree structures that are dependent on the number of versions provided per package. If only one version of a package is provided by CSE, then the layout in figure 2a or b can be used. If more than one version of a package is provide by CSE, then the layout in figure 2b must be used. Once the directory structure has been created, add the SConscript file `src.tar.gz` and the patch file if needed.

10.4.2 Generate Configuration Files for Scons

10.4.2.1 Generate a SConscript<sub:Generate-a-SConscript>. To illustrate how to create a SConscript file, consider the following SConscript file for the CSE release module Python with key components for creating a release CSE-compliant SConscript file:

- Lines 0–3: required in every SConscript file.
- Line 4: replace Python with the name of personal package.
- Line 5: replace 2.5.1 with the version of personal package.
- Lines 6–8: required in every SConscript file.
- Line 9: set release to 1 if the package is a release package; otherwise, set release to 0 (beta package).
- Line 10: set latest to 1 if the package is the latest version of the release package; otherwise, set release to 0 (all beta package require that release be set to 0).

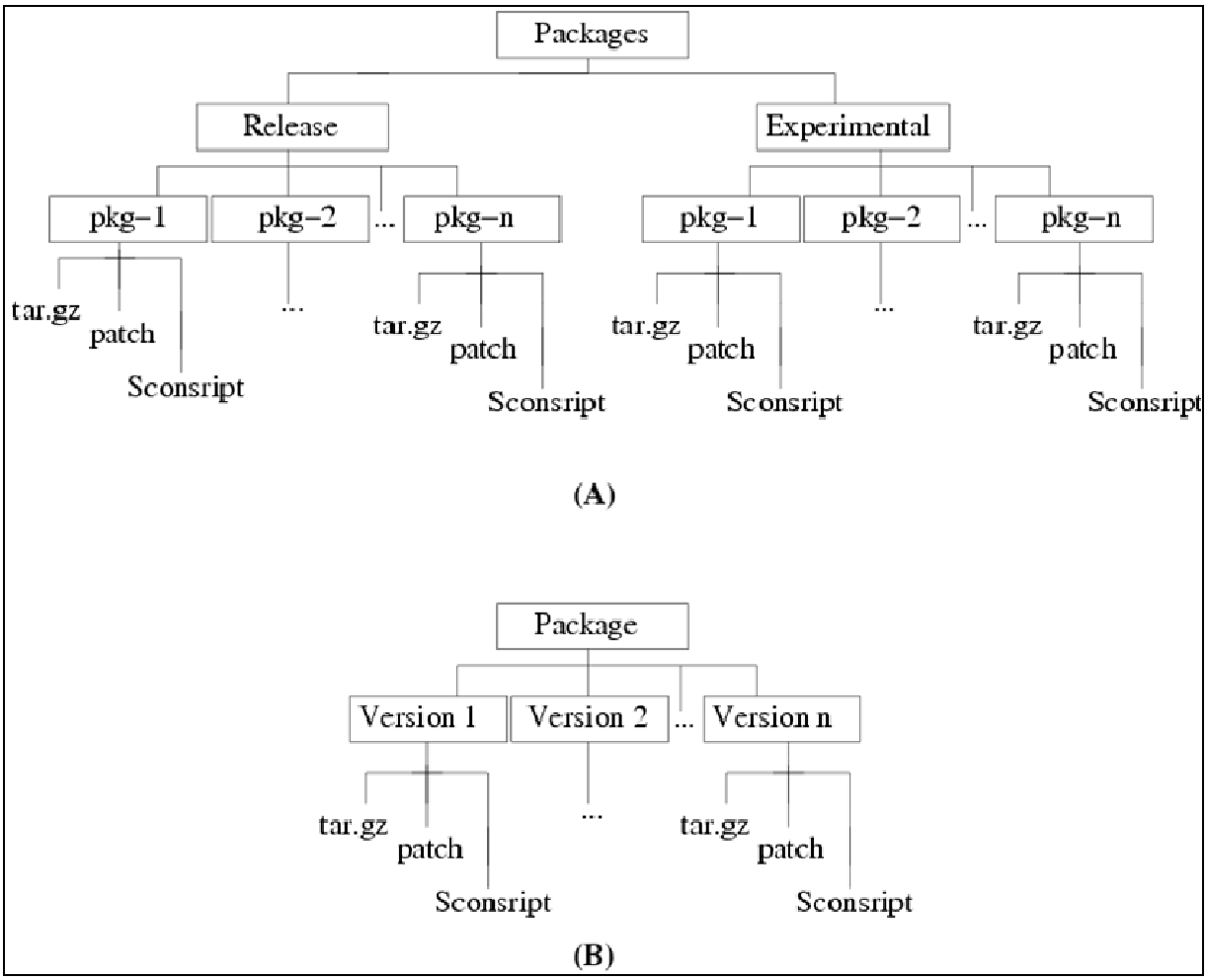

Figure 2. The CSE source package layout. (A) An overview of the general CSE package layout directory structure. (B) The directory layout corresponding to packages that have multiple versions.

- Line 11: add the list of dependent modules required by the package. To successfully create a module for package `_{\text{n}}`, all dependent module packages must be included in the module = `[('package_{\text{x}}','sys'),('package_{\text{y}} ','build'),('package_{\text{z}}','required')] list.`
  - sys: The sys flag indicates that a package _{\text{x}}module load will be queried. If package _{\text{x}}is loaded, then package _{\text{x}}is considered a dependent module.
  - build: The build flag indicates that package `_{\text{y}}` must be loaded to build and package `_{\text{y}}`is not a runtime dependency. If the package `_{\text{y}}` is not loaded, an attempt to load a CSE version of package `_{\text{y}}` is made. If package `_{\text{y}}` fails to load, the build stops with an error message indicating that it was unable to load package `_{\text{y}}`.

    - required: The required flag indicates that package _{\text{z}} must be loaded to build and package _{\text{z}} is a runtime dependency.
- Lines 12–21: required in every `SConscript` file.
- Line 22: add this line if a patch must be applied.
- Lines 23–25: add the correct build and install routines; recommend reviewing other SConscript files for examples.
- Lines 26–30: add any extra environment commands that are specific to the packages.
- Lines 31–37: required in every SConscript file.

Note that a SConscript file may vary from package to package and is slightly different for beta packages.

10.4.2.2 Update Main SConstruct. To make the CSE build aware of a custom package, add the following line to the main SConstruct (found in the CSE root source directory):

```
Release_Packages
+=[('myPackage','Packages/Release/myPackage/SConscript')]
```

or for beta packages:

```
Experimental_Packages +=[(
‘myPackage’,’Packages/Experimental/myPackage/SConscript’)].
```

The location of this additional line will become apparent when viewing the main CSE SConstruct file.

10.4.3 Patching a CSE Package

In some cases, it is required to patch a CSE file. To patch a CSE file:

1. Create a mod_src directory in the CSE build directory. The CSE build directory is generated after running a CSE build.
2. Copy the package modified src directory into the mod_src directory.
3. Remove the package src directory.
4. Modify SConscript line 20 of Section [sub:Generate-a-SConscript] to reflect the file or files that have been changed; recommend reviewing other SConscript files for examples.
5. Run a CSE build for the specified package to create a patch:

   ```
   > ./bootstrap -a mypackage diff=1
   ```

6. Remove the package src directory.

7. Run a final CSE build of package:

```
> ./bootstrap -a mypackage
```

---

11. CSE Helpful References

11.1 Subversion Repository

The CSE repository is stored on patrick using the Subversion (SVN) version control system. Additional information on Subversion can be found at http://svnbook.red-bean.com/.

11.1.1 Getting CSE Source

1. Load the CSE svn module:

```
> module load cse-tools
```

```
> module load cse/subversion
```

2. Get the latest version of CSE from SVN repository:

```
> svn co
```

```
svn+ssh://<username>@patrick/data/Repository/CSE/trunk/CSE SVN_CSE
```

(Note: The name of the directory SVN_CSE is arbitrary. If SVN_CSE does not exist, then one will be created. Also, in some cases, one may have to use patrick@arl.army.mil instead of only patrick.)

11.1.2 Export a Clean Directory Tree

To export a clean directory tree from the working copy:

1. Load the CSE svn module.
2. Go to SVN root directory or subdirectory.
3. Export a copy:

```
> svn export ./ DirName_CSE
```

(Note: The name of the directory DirName_CSE is arbitrary.)

11.1.3 Updating CSE Source

11.1.3.1 Update a File That Already Exist in CSE “Subversion” Repository.

1. Load the CSE subversion module.
2. Go to SVN root directory or subdirectory.

3. To update a file already in CSE SVN repository:

```
> svn update filename
```

(Note: This will mark the file as updated.)

4. Commit your changes:

```
> svn commit filename
```

(Note: This will open a text editor for you to record your comments. Save and exit editor to finish commit process.)

11.1.3.2 Add a New File to CSE SVN Repository

1. Load the CSE svn module.
2. Go to SVN root directory or subdirectory.
3. To add a file to CSE SVN repository:

```
> svn add filename
```

(Note: This will mark the file as updated.)

4. Commit your changes:

```
> svn commit filename
```

(Note: This will open a text editor for you to record your comments. Save and exit editor to finish commit process.)

11.1.3.3 Update Your CSE SVN Files

1. Load the CSE svn module.
2. Go to SVN root directory or subdirectory.
3. To update your SVN files in the current directory and subdirectories:

```
> svn update
```

11.2 Module Commands

- module avail: lists all available modules.
- module list: lists the all modules currently loaded.
- module load module_name: loads the module module_name.
- module add module_name: loads the module module_name.
- module unload module_name: unloads the module module_name.

- module rm module_name: unloads the module module_name.

11.3 LSF Commands

- bjobs: displays information about jobs.
- bkill: sends signals to kill jobs.
- bsub: submits a batch job to LSF.

Bibliography

Clarke, J. A.; Namburu, R. R. A Distributed Computing Environment for Interdisciplinary Applications. *Concurrency and Computation: Practice and Experience* **2002** *14*, 1161–1174.

Furlani, J. L. Modules: Providing a Flexible User Environment. In *Proceedings of the Fifth Large Installation Systems Administration Conference (LISA V)*, San Diego, CA, 30 September–3 October 1991.

Furlani, J. L.; Osel, P. W. Abstract Yourself With Modules, In LISA 96: *Proceedings of the 10th USENIX Conference on System Administration*, Berkeley, CA, 29 September–04 October 1996.

Kitware. *The ParaView Guide: A Parallel Visualization Application*; Clifton Park, NY, 2008.

Python. See URL: http://www.python.org.

Schroeder, W.; Martin, K.; Lorensen, B. *Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*, 4th ed.; Kitware: Clifton Park, NY, 2006.

NO. OF
COPIES ORGANIZATION

1 (PDF only) DEFENSE TECHNICAL INFORMATION CTR
DTIC OCA
8725 JOHN J KINGMAN RD
STE 0944
FORT BELVOIR VA 22060-6218

1 DIRECTOR
US ARMY RESEARCH LAB
IMNE ALC HRR
2800 POWDER MILL RD
ADELPHI MD 20783-1197

1 DIRECTOR
US ARMY RESEARCH LAB
RDRL CIM L
2800 POWDER MILL RD
ADELPHI MD 20783-1197

1 DIRECTOR
US ARMY RESEARCH LAB
RDRL CIM P
2800 POWDER MILL RD
ADELPHI MD 20783-1197

ABERDEEN PROVING GROUND

1 DIR USARL
RDRL CIM G (BLDG 4600)

INTENTIONALLY LEFT BLANK.