Hands-on session: Library and Application Deployment on Linux HPC Clusters with Environment Modules

Step 1: Import Virtual Machine

Create a directory for your virtual machine on the local disk as `/scratch/myvm`. Open the Virtual Box Software. Go to File->Preferences->General and change the default Virtual Machine folder `/scratch/myvm`. Then go to the File->Import Appliance dialog and import the `CentOS-6.4-64-bit.ova` pre-configured virtual machine. You should now be able to launch this virtual machine and log into it as superuser.

The root password for the virtual machine is: `h4nd$0FF`

Step 2: Create Installation Account and Installation Tree

Application and library software installation should never be done as superuser, thus we first create a user “install” that will specifically handle installation. After logging in as root type:

```
adduser --system --create-home --user-group install
chfn -f “Installation User” install
```

As next step we prepare a directory tree for software installation, preferably a location that can be easily exported to the entire cluster via a (read-only) NFS export or a local directory that can be kept synchronized to a master tree via rsync.

```
mkdir -p /opt/soft
chown install.install /opt/soft
chmod 0775 /opt/soft
```

Step 3: Compilation and Installation of the Modules Software

Switch to the installation user and create a directory for compilation, change into this directory, unpack the modules software sources, configure, compile and install it:

```
su -l install
mkdir compile
cd compile
tar -xzvvf /opt/sources/modules-3.2.10.tar.gz
cd modules-3.2.10
./configure --prefix=/opt/soft
make
make install
cd /opt/soft/Modules
ln -s 3.2.10 default
```

To activate the modules software, two files `modules.csh` and `modules.sh` need to be added to `/etc/profile.d`, so that the module command becomes available in interactive shells and shell scripts. Examples for both are below:

```
# for c-shells: /etc/profile.d/modules.csh
if ($?tcsh) then
        set modules_shell="tcsh"
else
        set modules_shell="csh"
endif

if ( -f /opt/soft/Modules/default/init/${modules_shell} ) then
   source /opt/soft/Modules/default/init/${modules_shell}
endif

setenv MODULERCFILE /opt/soft/modulerc
module add null

unset modules_shell
```

```
# for bourne-like shells /etc/profile.d/modules.sh
trap "" 1 2 3

MID=/opt/soft/Modules/default/init/

case "$0" in
    -bash|bash|*/bash) test -f $MID/bash && . $MID/bash ;;
       -ksh|ksh|*/ksh) test -f $MID/ksh  && . $MID/ksh ;;
          -sh|sh|*/sh) test -f $MID/sh   && . $MID/sh ;;
                    *) test -f $MID/sh   && . $MID/sh ;;
# default for scripts
esac

MODULERCFILE=/opt/soft/modulerc
export MODULERCFILE
module add null

trap - 1 2 3
```

In addition we need to generate the global modulerc file which is pointed to by `$MODULERCFILE`; this in our case we need to generate this file as `/opt/soft/modulerc`:

```
#%Module
append-path MODULEPATH /opt/soft/libs/modulefiles
append-path MODULEPATH /opt/soft/tools/modulefiles

# system-wide pre-loaded standard modules:
set defmodules {}
foreach m $defmodules {
  if {! [ is-loaded $m ] } {
    module load $m
  }
```

```
}
```

Step 4: Installing FFTW-3

For libraries deployed via environment modules, it is generally not desirable to build them as shared libraries, since they will require loading the (exact) same module at run time as well, which can easily turn into a maintenance nightmare. If shared libraries are unavoidable, it is usually best to bundle the matching version as a copy with the software that requires it. Here we build FFTW as a library module:

```
su -l install
cd compile
tar -xzvvf /opt/sources/fftw-3.3.3.tar.gz
cd fftw-3.3.3
./configure --prefix=/opt/soft/libs/fftw-3.3.3 --disable-shared \
     --enable-static --enable-single --enable-fortran
make
make install
make clean
./configure --prefix=/opt/soft/libs/fftw-3.3.3 --disable-shared \
     --enable-static --enable-fortran
make
make install
```

After compiling and installing the library, we need to write a module file with the required settings. It is usually beneficial to group custom installed software into logical units and use separate directory and module trees for them. The name of each module is a directory in the module path directory and for each version we create a file with the version number.

```
mkdir -p /opt/soft/libs/modulefiles/fftw
```

Here is an example for a module file for FFTW. Please see the modules documentation for details. This file would be stored as `/opt/soft/libs/modulefiles/fftw/3.3.3`:

```
#%Module###################################################
set version 3.3.3

proc ModulesHelp { } {
  puts stderr "This module provides the FFTW-3 library for calculating"
  puts stderr "the discrete Fourier transform in one or more dimensions.”
  puts stderr “It updates the environment variables \$CPATH, \$MANPATH,”
  puts stderr “\$LIBRARY_PATH and \$INFOPATH accordingly."
  puts stderr ""
  puts stderr "The following variables are defined for use in Makefiles:"
  puts stderr ""
  puts stderr "\$FFTW3_DIR, \$FFTW3_BIN, \$FFTW3_INC, \$FFTW3_LIB"
  puts stderr ""
}

module-whatis "FFTW-3 fast Fourier transform numerical library"

set prefix  /opt/soft/libs/fftw-${version}
prepend-path    CPATH           ${prefix}/include
```

```
prepend-path    LIBRARY_PATH    ${prefix}/lib
prepend-path    MANPATH         ${prefix}/share/man
prepend-path    INFOPATH        ${prefix}/share/info

setenv FFTW3_DIR  ${prefix}
setenv FFTW3_BIN  ${prefix}/bin
setenv FFTW3_LIB  ${prefix}/lib
setenv FFTW3_INC  ${prefix}/include
```

In addition to the per version module files, we can also set up a `.modulerc` file in the `fftw` directory, which can contain global settings for all fftw modules, e.g. some aliases and defaults.

```
#%Module
# set version aliases and defaults.
if {![info exists fftw-done]} {
  module-version fftw/3.3.3   3.3
  module-version fftw/3.3   default
  set fftw-done 1
}
```

Step 5: MPC Library (a Prerequisite for Building GCC)

Now build a module for the `mpc-0.8.2.tar.gz` package just like for FFTW. This library is required to build GCC 4.8.2 later on. The steps are the same and it is definitely desirable to have only a static version of this library installed. On newer Linux distribution this is also available through the distribution package manager, but for CentOS 6.x (and correspondingly RHEL 6.x) it is not directly available (only through the EPEL add-on repository).

Step 6: Two Concurrent Versions of OpenMPI

One of the main advantages of using environment modules is the option to have multiple concurrent versions of the same software installed and making switching between them convenient for the user. In addition one can also use the module scripts to define modules that cannot be completely unloaded, only swapped. We practice this with two concurrent installations of OpenMPI:

```
su -l install
cd compile
tar -xjvvf /opt/sources/openmpi-1.6.5.tar.bz2
cd openmpi-1.6.5
./configure --prefix=/opt/soft/tools/openmpi-1.6.5
make
make install
```

Now we set up a module file for OpenMPI:

```
mkdir -p /opt/soft/tools/modulefiles/openmpi
```

```
#%Module#####################################################################
## OpenMPI Message Passing Interface package
set version 1.6.5

proc ModulesHelp { } {
```

```
  puts stderr "This module enables using message passing interface libraries"
  puts stderr "of the OpenMPI distribution. The environment variables \$PATH,"
  puts stderr " \$LD_LIBRARY_PATH, and \$MANPATH accordingly."
  puts stderr "This version includes support for MPI threads."
  puts stderr ""
  puts stderr "The following variables are defined for use in Makefiles:"
  puts stderr "\$MPI_DIR, \$MPI_BIN, \$MPI_INC, \$MPI_LIB, \$MPI_FORTRAN_MOD_DIR"
  puts stderr " "
}

module-whatis "OpenMPI message passing interface package"

set prefix  /opt/soft/tools/openmpi-${version}

if { [module-info mode remove] && !([module-info mode switch1] \
     || [module-info mode switch3]) } {
  puts stderr "Module [module-info name] must not be unloaded."
  puts stderr "Use \"module switch\" if you need to replace it."
  break
}

prepend-path   PATH            ${prefix}/bin
prepend-path   LD_LIBRARY_PATH ${prefix}/lib
prepend-path   MANPATH         ${prefix}/share/man/
setenv         MPI_BIN         ${prefix}/bin
setenv         MPI_SYSCONFIG   ${prefix}/etc
setenv         MPI_FORTRAN_MOD_DIR ${prefix}/lib
setenv         MPI_INC         ${prefix}/include
setenv         MPI_LIB         ${prefix}/lib
setenv         MPI_MAN         ${prefix}/share/man
setenv         MPI_HOME        ${prefix}
```

As a special bonus, the way how OpenMPI is set up allows to use the same, gcc-compiled version also for other compilers, e.g. the Intel compiler suite. Here are the steps to add wrappers for them.

```
cd /opt/soft/tools/openmpi-1.6.5/bin
ln -s opal_wrapper mpiicc
ln -s opal_wrapper mpiicpc
ln -s opal_wrapper mpiifort
cd /opt/soft/tools/openmpi-1.6.5/share/openmpi
cp mpicc-wrapper-data.txt mpiicc-wrapper-data.txt
cp mpic++-wrapper-data.txt mpiicpc-wrapper-data.txt
cp mpif77-wrapper-data.txt mpiifort-wrapper-data.txt
```

In the *-wrapper-data.txt files replace `gcc` with `icc`, `g++` with `icpc`, and `gfortran` with `ifort`. In the latter file, also -pthread has to be removed to silence a harmless warning. After these changes the new wrappers will redirect compilation to the corresponding Intel compilers instead of the GCC suite.

Now install a second version of OpenMPI:

```
su -l install
cd compile
tar -xjvvf /opt/sources/openmpi-1.7.3.tar.bz2
```

```
cd openmpi-1.7.3
./configure --prefix=/opt/soft/tools/openmpi-1.7.3
make
make install
```

And set up its module file. This is pretty straightforward, with the example module file from above, which can be copied and then requires only the version variable to be edited.

```
cp /opt/soft/tools/modulefiles/openmpi/1.6.5 \
     /opt/soft/tools/modulefiles/openmpi/1.7.3
```

Since it is likely that there will be multiple compatible “patchlevel releases” for both OpenMPI version series, we define aliases for both of them and make the “stable” series the default in the .modulerc file.

```
#%Module
if {![info exists openmpi-done]} {
  module-version openmpi/1.6.5   1.6
  module-version openmpi/1.7.3   1.7
  module-version openmpi/1.6      default
  set openmpi-done 1
}
```

Finally we load the openmpi module by default in the global configuration in /opt/soft/modulerc

```
#%Module
append-path MODULEPATH /opt/soft/libs/modulefiles
append-path MODULEPATH /opt/soft/tools/modulefiles

# system-wide pre-loaded standard modules:
set defmodules {openmpi}
foreach m $defmodules {
  if {! [ is-loaded $m ] } {
    module load $m
  }
}
```

Step 7: Provisioning Custom GCC Compiler Versions

Compiling a compiler is quite time consuming and can be intimidating to less experienced users. Same as with libraries, in a cluster environment, it is desirable to minimize the dependencies on shared libraries. While with commercial compilers like the Intel compiler suite, this is not easily possible, with GCC, this is an available choice. On x86 machines, there is the added complication, that GCC will usually compile itself for 32-bit and 64-bit mode by default, and for the former, there is no assembler, header and runtime support available. Compilers need to be built in multiple stages with the final compiler being compiled by itself and tested to produce exact identical binaries while compiling itself. To ease this process, the compilation should be performed in a build tree separate from the sources. Finally, we only want to provide the custom compiler for the three most common languages: C, C++ and Fortran.

```
su -l install
cd compile
```

```
module load mpc
tar -xjvvf /opt/sources/gcc-4.8.2.tar.bz2
mkdir build-gcc
cd build-gcc
../gcc-4.8.2/configure --prefix=/opt/soft/tools/gcc-4.8.2    \
     --disable-shared --disable-multilib --with-tune=generic \
     --enable-languages=c,c++,fortran --enable-threads=posix
make
make install
```

This step will take quite a while, since the compiler will be compiled multiple times including various run time library packages and support for multiple programming languages.

A suitable module file could look like this:

```
%Module######################################################################

set version 4.8.2

proc ModulesHelp { } {
  puts stderr "This module provides an version ${version} of the GNU C,”
  puts stderr "C++ and Fortran compilers as an alternative to the operating”
  puts stderr "system default version 4.4.6 GNU compilers.\n"
  puts stderr ""
}

module-whatis "Alternate GNU C/C++/Fortran Compilers"

set prefix  /opt/tools/gcc-${version}

prepend-path    PATH            ${prefix}/bin
prepend-path    MANPATH         ${prefix}/man
prepend-path    LIBRARY_PATH    ${prefix}/lib64
```

Step 8: Compiling an Application like Quantum Espresso

To compile Quantum Espresso we can now utilize several of the modules that we have built during the previous steps: `openmpi`, `fftw`, and `gcc`. For testing purposes we are compiling a development snapshot. Modules offer a convenient way to make these available to users with a specific need without enabling them by default for all users.

```
su -l install
cd compile
tar -xzvvf /opt/sources/espresso-r10544.tar.gz
cd espresso-r10544
module purge
module load openmpi
module load gcc
module load fftw
./configure
make all
```

Quantum Espresso has no install target like other typical source packages and most users that want a precompiled package only require the executables. So we just copy the binaries manually to a directory

```
mkdir -p /opt/soft/tools/espresso-r10544
cp bin/*.x /opt/soft/tools/espresso-r10544/
mkdir -p /opt/soft/tools/modulefiles/espresso
```

and write a minimal module file for the Quantum ESPRESSO binaries

```
%Module######################################################################

module-whatis "Development Snapshot of the Quantum ESPRESSO Package”

prepend-path    PATH  /opt/tools/espresso-r10544
```