Sooner or later every cluster develops a plethora of tools and libraries for applications or for building applications. Often the applications or tools need different compilers or different MPI libraries, so how do you handle situations in which you need to change tool sets or applications? You can do it the hard way, or you can do it the easy way with Environment Modules.

Environment Modules – A Great Tool for Clusters

Jeff Layton

When people first start using clusters, they tend to stick with whatever compiler and MPI library came with the cluster when it was installed. As they become more comfortable with the cluster, using the compilers, and using the MPI libraries, they start to look around at other options: Are there other compilers that could perhaps improve performance? Similarly, they might start looking at other MPI libraries: Can they help improve performance? Do other MPI libraries have tools that can make things easier? Perhaps even more importantly, these people would like to install the next version of the compilers or MPI libraries so they can test them with their code. So this forces a question: How do you have multiple compilers and multiple MPI libraries on the cluster at the same time and not get them confused? I'm glad you asked.

The Hard Way

If you want to change you compiler or libraries – basically anything to do with your environment – you might be tempted to change your *$PATH* in the *.bashrc* file (if you are using Bash) and then log out and log back in whenever you need to change your compiler/MPI combination. Initially this sounds like a pain, and it is, but it works to some degree. It doesn't work in the situation where you want to run multiple jobs each with a different compiler/MPI combination.

For example, say I have a job using the GCC 4.6.2 compilers using Open MPI

1.5.2, then I have a job using GCC 4.5.3 and MPICH2. If I have both jobs in the queue at the same time, how can I control my *.bashrc* to make sure each job has the correct *$PATH*? The only way to do this is to restrict myself to one job in the queue at a time. When it's finished I can then change my *.bashrc* and submit a new job. Because you are using a different compiler/MPI combination from what is in the queue, even for something as simple as code development, you have to watch when the job is run to make sure your *.bashrc* matches your job.

The Easy Way

A much better way to handle compiler/MPI combinations is to use Environment Modules. (Be careful not to confuse "environment modules" with "kernel modules.") According to the website, "The Environment Modules package provides for the dynamic modification of a user's environment via modulefiles." Although this might not sound earth shattering, it actually is a quantum leap for using multiple compilers/MPI libraries, but you can use it for more than just that, which I will talk about later.

You can use Environment Modules to alter or change environment variables such as *$PATH*, *$MANPATH*, *$LD_LIBRARY_LOAD*, and others. Because most job scripts for resource managers, such as LSF, PBS-Pro, and MOAB, are really shell scripts, you can incorporate Environment Modules into the scripts to set the appropriate *$PATH* for your compiler/MPI combination, or any other environment variables an application requires for operation.

How you install Environment Modules depends on how your cluster is built. You can build it from source, as I will discuss later, or you can install it from your package manager. Just be sure to look for Environment Modules.

Using Environment Modules

To begin, I'll assume that Environment Modules is installed and functioning correctly, so you can now test a few of the options typically used. In this article, I'll be using some examples from TACC. The first thing to check is what modules are available to you by using the *module avail* command:

```
[laytonjb@dlogin-0 ~]$ module avail

------------------------------------------ /opt/apps/intel11_1/
fftw3/3.2.2    gotoblas2/1.08 hdf5/1.8.4     mkl/10.2.4.032 mvap

------------------------------------------------ /opt/apps/modul
gnuplot/4.2.6       intel/11.1(default) papi/3.7.2
intel/10.1          lua/5.1.4           pgi/10.2

------------------------------------------------- /opt/modulef
Linux      TACC       TACC-paths cluster
------------------------------------------------ /cm/shared/modul
acml/gcc/64/4.3.0                  fftw3/gcc/64/3.2.2
acml/gcc/mp/64/4.3.0               fftw3/open64/64/3.2.2
acml/gcc-int64/64/4.3.0            gcc/4.3.4
```

```
acml/gcc-int64/mp/64/4.3.0         globalarrays/gcc/openmpi/64/
acml/open64/64/4.3.0               globalarrays/open64/openmpi/
acml/open64-int64/64/4.3.0         hdf5/1.6.9
blacs/openmpi/gcc/64/1.1patch03    hpl/2.0
blacs/openmpi/open64/64/1.1patch03 intel-cluster-checker/1.3
blas/gcc/64/1                      intel-cluster-runtime/2.1
blas/open64/64/1                   intel-tbb/ia32/22_20090809os
bonnie++/1.96                      intel-tbb/intel64/22_2009080
cmgui/5.0                          iozone/3_326
default-environment                lapack/gcc/64/3.2.1
fftw2/gcc/64/double/2.1.5          lapack/open64/64/3.2.1
fftw2/gcc/64/float/2.1.5           mpich/ge/gcc/64/1.2.7
fftw2/open64/64/double/2.1.5       mpich/ge/open64/64/1.2.7
fftw2/open64/64/float/2.1.5        mpich2/smpd/ge/gcc/64/1.1.1p
```

This command lists what environment modules are available. You'll notice that TACC has a very large number of possible modules that provide a range of compilers, MPI libraries, and combinations. A number of applications show up in the list as well.

You can check which modules are "loaded" in your environment by using the *list* option with the *module* command:

```
[laytonjb@dlogin-0 ~]$ module list
Currently Loaded Modulefiles:
  1) Linux          2) intel/11.1     3) mvapich2/1.4   4) sge/
```

This indicates that when I log in, I have six modules already loaded for me. If I want to use any additional modules, I have to load them manually:

```
[laytonjb@dlogin-0 ~]$ module load gotoblas2/1.08
[laytonjb@dlogin-0 ~]$ module list
Currently Loaded Modulefiles:
  1) Linux            3) mvapich2/1.4     5) cluster          7)
  2) intel/11.1       4) sge/6.2u3        6) TACC
```

You can just cut and paste from the list of available modules to load the ones you want or need. (This is what I do, and it makes things easier.) By loading a module, you will have just changed the environmental variables defined for that module. Typically this is *$PATH*, *$MANPATH*, and *$LD_LIBRARY_LOAD*.

To unload or remove a module, just use the *unload* option with the *module* command, but you have to specify the complete name of the environment module:

```
[laytonjb@dlogin-0 ~]$ module unload gotoblas2/1.08
[laytonjb@dlogin-0 ~]$ module list
Currently Loaded Modulefiles:
  1) Linux          2) intel/11.1     3) mvapich2/1.4   4) sge/
```

Notice that the *gotoblas2/1.08* module is no longer listed. Alternatively, to you can

unload all loaded environment modules using *module purge*:

```
[laytonjb@dlogin-0 ~]$ module purge
[laytonjb@dlogin-0 ~]$ module list
No Modulefiles Currently Loaded.
```

You can see here that after the *module purge* command, no more environment modules are loaded.

If you are using a resource manager (job scheduler), you are likely creating a script that requests the resources and runs the application. In this case, you might need to load the correct Environment Modules in your script. Typically after the part of the script in which you request resources (in the PBS world, these are defined as *#PBS* commands), you will then load the environment modules you need.

Now that you've seen a few basic commands for using Environment Modules, I'll go into a little more depth, starting with installing from source. Then I'll use the module in a job script and write my own module.

Building Environment Modules for Clusters

In my opinion, the quality of open source code has improved over the last several years to the point at which building and installing is fairly straightforward, even if you haven't built any code before. If you haven't built code, don't be afraid to start with Environment Modules.

For this article, as an example, I will build Environment Modules on a "head" node in the cluster in */usr/local*. I will assume that you have */usr/local* NSF exported to the compute nodes or some other filesystem or directory that is mounted on the compute nodes (perhaps a global filesystem?). If you are building and testing your code on a production cluster, be sure to check that */usr/local* is mounted on all of the compute nodes.

To begin, download the latest version – it should be a **.tar.gz* file. (I'm using v3.2.6, but the latest as of writing this article is v3.2.9). To make things easier, build the code in */usr/local*. The documentation that comes with Environment Modules recommends that it be built in */usr/local/Modules/src*. As root, run the following commands:

```
% cd /usr/local
% mkdir Modules
% cd Modules
% mkdir src
% cp modules-3.2.6.tar.gz /usr/local/Modules/src
% gunzip -c modules-3.2.6.tar.gz | tar xvf -
% cd modules-3.2.6
```

At this point, I would recommend you carefully read the *INSTALL* file; it will save your bacon. (The first time I built Environment Modules, I didn't read it and had lots of trouble.)

Before you start configuring and building the code, you need to fulfill a few

prerequisites. First, you should have Tcl installed, as well as the Tcl Development package. Because I don't know what OS or distribution you are running, I'll leave to you the tasks of installing Tcl and Tcl Development on the node where you will be building Environment Modules.

At this point, you should configure and build Environment Modules. As root, enter the following commands:

```
% cd /usr/local/Modules/src/modules-3.2.6
% ./configure
% make
% make install
```

The *INSTALL* document recommends making a symbolic link in */usr/local/Modules* connecting the current version of Environment Modules to a directory called *default*:

```
% cd /usr/local/Modules
% sudo ln -s 3.2.6 default
```

The reason they recommend using the symbolic link is that, if you upgrade Environment Modules to a new version, you build it in */usr/local/Modules/src* and then create a symbolic link from */usr/local/Modules/<new>* to */usr/local/Modules /default*, which makes it easier to upgrade.

The next thing to do is copy one (possibly more) of the init files for Environment Modules to a global location for all users. For my particular cluster, I chose to use the *sh* init file. This file will configure Environment Modules for all of the users. I chose to use the *sh* version rather than *csh* or *bash*, because *sh* is the least common denominator:

```
% sudo cp /usr/local/Modules/default/init/sh /etc/profile.d/modu
% chmod 755 /etc/profile.d/modules.sh
```

Now users can use Environment Modules by just putting the following in their *.bashrc* or *.profile*:

```
%. /etc/profile.d/modules.sh
```

As a simple test, you can run the above script and then type the command *module*. If you get some information about how to use modules, such as what you would see if you used the *-help* option, then you have installed Environment Modules correctly.

Environment Modules in Job Scripts

In this section, I want to show you how you can use Environment Modules in a job script. I am using PBS for this quick example, with this code snippet for the top part of the job script:

```
#PBS -S /bin/bash
```

```
#PBS -l nodes=8:ppn=2

. /etc/profile.d/modules.sh
module load compiler/pgi6.1-X86_64
module load mpi/mpich-1.2.7

(insert mpirun command here)
```

At the top of the code snippet is the PBS directives that begin with *#PBS*. After the PBS directives, I invoke the Environment Modules startup script (*modules.sh*). Immediately after that, you should *load* the modules you need for your job. For this particular example, taken from a three-year-old job script of mine, I've loaded a compiler (*pgi 6.1-x86_64*) and an MPI library (*mpich-1.2.7*).

Building Your Own Module File

Creating your own module file is not too difficult. If you happen to know some Tcl, then it's pretty easy; however, even if you don't know Tcl, it's simple to follow an example to create your own.

The modules themselves define what you want to do to the environment when you load the module. For example, you can create new environment variables that you might need to run the application or change *$PATH*, *$LD_LIBRARY_LOAD*, or *$MANPATH* so a particular application will run correctly. Believe it or not, you can even run code within the module or call an external application. This makes Environment Modules very, very flexible.

To begin, remember that all modules are written in Tcl, so this makes them very programmable. For the example, here, all of the module files go in */usr/local /Modules/default/modulefiles*. In this directory, you can create subdirectories to better label or organize your modules.

In this example, I'm going to create a module for *gcc-4.6.2* that I build and install into my home account. To begin, I create a subdirectory called *compilers* for any module file that has to do with compilers. Environment Modules has a sort of template you can use to create your own module. I used this as the starting point for my module. As root, do the following:

```
% cd /usr/local/Modules/default/modulefiles
% mkdir compilers
% cp modules compilers/gcc-4.6.2
```

The new module will appear in the module list as *compilers/gcc-4.6.2*. I would recommend that you look at the template to get a feel for the syntax and what the various parts of the modulefile are doing. Again, recall that Environment Modules use Tcl as its language but you don't have to know much about Tcl to create a module file. The module file I created follows:

```
#%Module1.0#####################################################
##
## modules compilers/gcc-4.6.2
```

```
##
## modulefiles/compilers/gcc-4.6.2.  Written by Jeff Layton
##
proc ModulesHelp { } {
        global version modroot

        puts stderr "compilers/gcc-4.6.2 - sets the Environment
}

module-whatis   "Sets the environment for using gcc-4.6.2 compi

# for Tcl script use only
set     topdir          /home/laytonj/bin/gcc-4.6.2
set     version         4.6.2
set     sys             linux86

setenv          CC              $topdir/bin/gcc
setenv          GCC             $topdir/bin/gcc
setenv          FC              $topdir/bin/gfortran
setenv          F77             $topdir/bin/gfortran
setenv          F90             $topdir/bin/gfortran
prepend-path    PATH            $topdir/include
prepend-path    PATH            $topdir/bin
prepend-path    MANPATH         $topdir/man
prepend-path    LD_LIBRARY_PATH $topdir/lib
```

The file might seem a bit long, but it is actually fairly compact. The first section provides help with this particular module if a user asks for it (the line that begins with *puts stderr*); for example:

```
home8:~> module help compilers/gcc-4.6.2

----------- Module Specific Help for 'compilers/gcc-4.6.2' -----

compilers/gcc-4.6.2 - sets the Environment for GCC 4.6.2 in my l
```

You can have multiple strings by using several *puts stderr* lines in the module (the template has several lines).

After the help section in the procedure *ModuleHelp*, another line provides some simple information when a user uses the *whatis* option; for example:

```
home8:~> module whatis compilers/gcc-4.6.2
compilers/gcc-4.6.2  : Sets the environment for using gcc-4.6.2
```

After the *help* and *whatis* definitions is a section where I create whatever environment variables are needed, as well as modify *$PATH*, *$LD_LIBRARY_PATH*, and *$MANPATH* or other standard environment variables. To make life a little easier for me, I defined some local variables:*topdir*, *version*, and

sys. I only used *topdir*, but I defined the other two variables in case I needed to go back and modify the module (the variables can help remind me what the module was designed to do).

In this particular modulefile, I defined a set of environment variables pointing to the compilers (CC, GCC, FC, F77, and F90). After defining those environment variables, I modified *$PATH*, *$LD_LIBRARY_PATH*, and *$MANPATH* so that the compiler was first in these paths by using the *prepend-path* directive.

This basic module is pretty simple, but you can get very fancy if you want or need to. For example, you could make a module file dependent on another module file so that you have to load a specific module before you load the one you want. Or, you can call external applications – for example, to see whether an application is installed and functioning. You are pretty much limited only by your needs and imagination.

Making Sure It Works Correctly

Now that you've defined a module, you need to check to make sure it works. Before you load the module, check to see which *gcc* is being used:

```
home8:~> which gcc
/usr/bin/gcc
home8:~> gcc -v
Reading specs from /usr/lib/gcc/i386-redhat-linux/3.4.3/specs
Configured with: ../configure --prefix=/usr --mandir=/usr/share/
--infodir=/usr/share/info --enable-shared --enable-threads=posi
--disable-checking --with-system-zlib --enable-__cxa_atexit
--disable-libunwind-exceptions--enable-java-awt=gtk
--host=i386-redhat-linux Thread model: posix
gcc version 3.4.3 20050227 (Red Hat 3.4.3-22.1)
```

This means *gcc* is currently pointing to the system *gcc*. (Yes, this is a really old gcc; I need to upgrade my simple test box at home).

Next, load the module and check which *gcc* is being used:

```
home8:~> module avail

---------------------------- /usr/local/Modules/versions ------
3.2.6

------------------------- /usr/local/Modules/3.2.6/modulefiles -
compilers/gcc-4.6.2 dot                 module-info         null
compilers/modules   module-cvs          modules             use
home8:~> module load compilers/gcc-4.6.2
home8:~> module list
Currently Loaded Modulefiles:
  1) compilers/gcc-4.6.2
home8:~> which gcc
~/bin/gcc-4.6.2/bin/gcc
```

```
home8:~> gcc -v
Using built-in specs.
Target: i686-pc-linux-gnu
Configured with: ./configure --prefix=/home/laytonj/bin/gcc-4.6
--enable-languages=c,fortran --enable-libgomp
Thread model: posix
gcc version 4.6.2
```

This means if you used *gcc*, you would end up using the version built in your home directory.

As a final check, *unload* the module and recheck where the default *gcc* points:

```
home8:~> module unload compilers/gcc-4.6.2
home8:~> module list
No Modulefiles Currently Loaded.
home8:~> which gcc
/usr/bin/gcc
home8:~> gcc -v
Reading specs from /usr/lib/gcc/i386-redhat-linux/3.4.3/specs
Configured with: ../configure --prefix=/usr --mandir=/usr/share/
--infodir=/usr/share/info --enable-shared --enable-threads=posi
--disable-checking --with-system-zlib --enable-__cxa_atexit
--disable-libunwind-exceptions--enable-java-awt=gtk
--host=i386-redhat-linux
Thread model: posix
gcc version 3.4.3 20050227 (Red Hat 3.4.3-22.1)
```

Notice that after you unload the module, the default *gcc* goes back to the original version, which means the environment variables are probably correct. If you want to be more thorough, you should check all of the environment variables before loading the module, after the module is loaded, and then after the module is unloaded. But at this point, I'm ready to declare success!

Final Comments

For clusters, Environment Modules are pretty much the best solution for handling multiple compilers, multiple libraries, or even applications. They are easy to use even for beginners to the command line. Just a few commands allow you to add modules to and remove them from your environment easily. You can even use them in job scripts. As you also saw, it's not too difficult to write your own module and use it. Environment Modules are truly one of the indispensable tools for clusters.