SAND2020-6782C

Module Systems and Developer Productivity

A whitepaper for the [2020 Collegeville Workshop on Scientific Software](https://collegeville.github.io/CW20/),focusing on Developer Productivity.

[Joe Frye](mailto:jfrye@sandia.gov); [*Sandia National Laboratories*](www.sandia.gov)

Introduction

Many modern scientific software projects depend upon a common ecosystem of community software. It is not uncommon for a project to depend on a dozen different third-party libraries. These dependencies are often highly configurable, difficult to build, and can have tailored dependencies of their own. All too often, developers are forced to spend a significant amount of time just putting together a working environment in which to develop. That lost time that could have been spent on more valuable activities. Additionally, having developers install their own dependencies leads to inconsistent development environments across a team; that is, each developer may install slightly different versions of libraries with slightly different configurations. This frequently leads to defects and inconsistent behaviors and causes developers to spend an inordinate amount of time rooting out bugs because it is unclear whether the problems are in the application itself or in one of its numerous dependencies. Moreover, having to configure and build libraries puts an onerous tax on build and test cycles; I have personally been involved with projects where 75% of the build time is just building dependencies.

Providing developers with a common infrastructure to access the dependencies they need in a consistent, easy-to-use way can solve these problems and directly improves developer productivity. First, if a system can provide pre-installed, ready-made libraries, there is no wasted time spent by developers installing dependencies. Second, by provisioning a consistent set of packages across development environments, the system makes it easier for developers to locate defects in their project code. Finally, if libraries can be made available on-demand, then projects can build their code from a clean slate without added overhead. In short, this means that developers can get on with the business of development.

All of these things can be accomplished by having a sufficient and mature environment module system. In this white paper, we will describe the design and implementation decisions involved in constructing such a system, using a real-world example.

Background

By a module system, here we mean a collection of environment modules that have been curated in some way and that provides a standard way for users to access their dependencies. Environment modules are way to modify a user's environment to include some preinstalled software. Users can issue commands like `module load`, `module avail`, and `module list` to manipulate their environment, see what preinstalled software they can leverage, and see which modules they have already loaded. Environment modules at the basic level are just a mechanism to set and manipulate environment and variables; this allows the system to expose or hide pre-installed libraries. A module for a third-party library might set some environment variables to let the user know where those libraries are located as well as append that path to `LD_LIBRARY_PATH`. Meanwhile, an environment module for a utility would add the `bin/` directory to `PATH`. When a user unloads a module then all the environment variables that were set go back to being unset and any paths that were prepended to `LD_LIBRARY_PATH` and `PATH` are removed from these variables. Here is an example of loading and unloading a python module:

```console
$ which python
/usr/bin/python
$ module load python/3.7.3
$ which python
/home/projects/x86-64/python/3.7.3/bin/python
$ module unload python/3.7.3
$ which python
/usr/bin/python
```

In short, when we talk about a module system, we are referring to a collection of software packages that have been created in an intentional way to work together and are accessible using environment modules. In order to be most effective, a module system should have certain characteristics which we shall discuss in the following section.

Design Considerations

Using a well-thought-out environment module system has some significant advantages to developer productivity. Projects can greatly reduce build time so developers can see results much quicker because they do not have to build their dependencies in addition to their project's source code. This is especially pronounced in automated testing where it is common to build completely from scratch. Another advantage is that such a system can standardize the programming environment across the project. All the developers use the same version and configuration of the dependencies. Differences between two developers' versions of the project are almost certainly due to code in the project itself, not from the environment. Once again, by providing a common environment, such a module system can make reproducibility much easier between developers and platforms.

Easy to Use

Perhaps the most important thing to consider when designing a module system is that it needs to be easy for the user to see what is available and easy for the user to load a consistent set of software into their environment. This means that when they do a `module avail` they see reasonable output that is organized in such a way that they can easily tell what is available to them. I have seen many cases where modules are unorganized and follow no standard. This quickly becomes impossible to navigate for users. The following is an example of something I see often and hopefully illustrates the point that this can quickly become confusing:

```

openmpi/1.10
openmpi/1.10_manual
openmpi/1.10_update
openmpi/1.8
openmpi/1.8_intel-2018
openmpi/2.1.3
openmpi/2.1.3_gcc-4.9.3
openmpi/2.1.6_not_this_one
openmpi/2.1.6_with-hwloc
openmpi/2.1.6_with-pmix
```

Some of these give you detail about which compiler was used to build it, some of them give detail about some configure options that were used, some have an English word or phrase, and some are just a version. I can wager a reasonable guess at how they all differ but it is not entirely clear. Suppose I am new to this platform and I know I need openmpi 2 for my project it is not clear which to choose or why.

Consistency

Another important factor to consider while designing a module system is consistency. I think of this in a few ways in the context of modules. One is that all the modules in the system should behave in a similar well defined way that the users can easily predict. For example you may want to set some environment variables when a module is loaded to point to the install root of the package as well as where the libraries are in that install so a module for boost may set `BOOST_ROOT` and `BOOST_LIB_DIR`. Consistency in this case is that when you create a module for hdf5, you set the environment variables `HDF5_ROOT` and `HDF5_LIB_DIR` not `HDF5_BASE` and `HDF5_LIBRARIES`. It does not matter what you want to call the environment variables but pick something that is easy for users to predict. The other aspect of consistency that is helpful is consistency between platforms. Teams are expected to run on multiple platforms so as much as possible the modules systems should be similar on each different platform. A familiar and easy to use interface to get dependencies will cut down on frustration and increase developer productivity.

Implementation Decisions

Now we can delve into the challenges in realizing an effective solution for dependency management. There are many things to consider when implementing a module system For example, maintainers can install similar versions of a package and rarely have a strategy to deprecate them, which increases the maintenance burden and pollutes the system. However we will focus on the particular issue of making it easy for a user to load consistent modules.

How to Communicate which Modules are Consistent with Each Other

In order to ensure that TPLs work together they usually need to be built with the same compiler, mpi, and sometimes other dependencies. Suppose you are supporting 2 compilers, 2 mpis, and package "A" that needs mpi. You will need to build package "A" 4 separate times (one for each compiler/mpi combination). To make things more concrete, let's name the compilers and mpis. For purposes of the example let's say we are supporting gcc-7.2.0 and intel-19 with openmpi-3.0.0 and openmpi-4.0.0 and package "A" at version 1.0.0. To build our package "A" we will build once with gcc-7.2.0 and openmpi-3.0.0, once with gcc-7.2.0 and openmpi-4.0.0, and so on.

```mermaid
graph TD;
  gcc7[gcc-7.2.0] --> ompi3gcc7[openmpi-3.0.0 built with gcc-7.2.0];
  gcc7 --> ompi4gcc7[openmpi-4.0.0 built with gcc-7.2.0];
  I19[intel-19] --> ompi3I19[openmpi-3.0.0 built with Intel-19];
  I19 --> ompi4I19[openmpi-4.0.0 built with Intel-19];
  ompi3gcc7 --> Agcc7ompi3[Package A built with gcc-7.2.0 and openmpi-3.0.0]
  ompi4gcc7 --> Agcc7ompi4[Package A built with gcc-7.2.0 and openmpi-4.0.0]
  ompi3I19 --> AI199ompi3[Package A built with Intel-19 and openmpi-3.0.0]
  ompi4I19 --> AI19ompi4[Package A built with Intel-19 and openmpi-4.0.0]
```

How do you communicate to users which modules are compatible?
Use Long Module Names: One way this is done in a flat module system is to embed that information in the module name. so the users would see 4 modules for package A:

```

A/1.0.0/gcc/7.2.0/openmpi/3.0.0
A/1.0.0/gcc/7.2.0/openmpi/4.0.0
```

~~~~
A/1.0.0/intel/19.0.0/openmpi/3.0.0
A/1.0.0/intel/19.0.0/openmpi/4.0.0
```
The user is then expected to understand that
"A/1.0.0/gcc/7.2.0/openmpi/3.0.0" means that if you "module load
A/1.0.0/gcc/7.2.0/openmpi/3.0.0" you will get package "A" at version 1.0.0
added to your environment and that it will have been built with gcc-7.2.0 and
openmpi-3.0.0.  Of course, the example above only shows the modules for "A".
In this simple example we would need all of the following modules available
to users:
```
gcc/7.2.0
intel/19.0.0
openmpi/3.0.0/gcc/7.2.0
openmpi/3.0.0/intel/19.0.0
openmpi/4.0.0/gcc/7.2.0
openmpi/4.0.0/intel/19.0.0
A/1.0.0/gcc/7.2.0/openmpi/3.0.0
A/1.0.0/gcc/7.2.0/openmpi/4.0.0
A/1.0.0/intel/19.0.0/openmpi/3.0.0
A/1.0.0/intel/19.0.0/openmpi/4.0.0
```
In order to get a consistent set of dependencies a user then needs to:
```
module load gcc/7.2.0
module load openmpi/4.0.0/gcc/7.2.0
module load A/1.0.0/gcc/7.2.0/openmpi/4.0.0
```
making sure the compiler and MPI are consistent in what they load.  This is
obviously annoying and becomes tedious for users once you have more than a
couple supported packages.
**Create Devpacks**: One way to help is to create "devpacks", where a devpack
is a single module that loads other modules known to be consistent, or it
could also do all the work of putting the consistent set of software in a
user's environment in one module.  This makes it easy for users to load the
right set of modules to be consistent but it adds to the clutter of seeing
all the modules at once.
**Provide Smart Modules**: Another approach is to have modules that query the
environment and then set environment variables and paths based on what has
already been loaded.  In this type of system, you would only have one module
for any given version of a package, for example `boost/1.70.0`, but the
module would decide at load time which installation to point to based on
environment variables set by other modules.  In this way, the number of
modules displayed through `module avail` is dramatically decreased but users
can still see what software and versions are available through modules.  One
of the largest strengths of this style is that it presents a nice easy to use
interface for developers.
**Hierarchical Modules**: The final way to deal with this in my experience is
to use LUA-based hierarchical modules.  In this approach when a user does a
`module avail` the first time they only see compiler modules.  After loading
one of the compiler modules, `module avail` will display a new set of modules
that have all been built with the loaded compiler.  There can be multiple
levels in the hierarchy, another common one is MPI.  If this is the case then
a user will see a new set of moduels available after they load an mpi module.
 This ensures that the user loads a consistent set of modules but it is not
obvious exactly what is available on the system and the user needs to do some
digging to find what they need.
## Discussion
At Sandia we can find all of the above implementations on different on
different projects.  One in particular has been especially impactful which is
a smart module system where the installations and module files are shared
across machines.  These two things combine to make an especially useful
system that is easy to use and is widely available.  As mentioned above, the
smart modules greatly reduce the clutter a user sees by deciding which
installation to point to at load time based on what has already been loaded.
This system clearly presents what modules a user can load and is easy to use.
 By having one install that is shared we can add software for customers just
by installing in one place.  Additionally, if defects are discovered we can
address them once, and the changes will propagate to all the other users.
```mermaid
graph LR;
  modules[Module System] --> share[modules and installations shared on the
network];
  installs[Install tree] --> share;
  share --> WS1[Workstation]
  share --> WS2[Workstation]
  share --> WS3[Workstation]
  share --> Tester1[Shared Project Resource]
  share --> Tester2[CI Build Machine]
  share --> Tester3[Nightly Testing Machine]
```
~~~~

This module system is shared with roughly one hundred machines across dozens of projects. This has reduced the time that developers spend trying to get a consistent working set of dependencies because individual developers no longer need to maintain their own TPL stack. This has also proven valuable in getting new developers spun up faster. Using this module system has served to standardize the dependency stack for projects that use it. Now developers are using the same environment as each other and as the testing infrastructure. This had lead to productivity gains through increased reproducibility of defects making them easier to resolve.

Conclusion

Scientific software projects often rely on a complex set of dependencies that can be very difficult to build and maintain. Providing developers with a common infrastructure to access the dependencies they need in a consistent, easy-to-use way can solve several problems that directly impact developer productivity. Such a system lessens the burden on developers for maintaining the TPL stack, standardizes the environment with testing resources, increases reproducibility through standardization, and allows developers to spend more time developing on their projects.

Acknowledgements

This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energyâ€™s National Nuclear Security Administration under contract DE-NA0003525.