



Building Software Environments for Research Computing Clusters

Mark Howison, Aaron Shen, and Andrew Loomis, *Brown University*

<https://www.usenix.org/conference/lisa13/technical-sessions/papers/howison>

This paper is included in the Proceedings of the
27th Large Installation System Administration Conference (LISA '13).

November 3–8, 2013 • Washington, D.C., USA

ISBN 978-1-931971-05-8

Open access to the
Proceedings of the 27th Large Installation
System Administration Conference (LISA '13)
is sponsored by USENIX.

Building Software Environments for Research Computing Clusters

Mark Howison

Brown University

Aaron Shen

Brown University

Andrew Loomis

Brown University

Abstract

Over the past two years, we have built a diverse software environment of over 200 scientific applications for our research computing platform at Brown University. In this report, we share the policies and best practices we have developed to simplify the configuration and installation of this software environment and to improve its usability and performance. In addition, we present a reference implementation of an environment modules system, called PyModules, that incorporates many of these ideas.

Tags

HPC, software installation, configuration management

1 Introduction

Universities are increasingly centralizing their research compute resources from individual science departments to a single, comprehensive service provider. At Brown University, that provider is the Center for Computation and Visualization (CCV), and it is responsible for supporting the computational needs of users from over 50 academic departments and research centers including the life, physical, and social sciences. The move to centralized research computing has created an increasing demand for applications from diverse scientific domains, which have diverse requirements, software installation procedures and dependencies. While individual departments may need to provide only a handful of key applications for their researchers, a service provider like CCV may need to support hundreds of applications.

At last year's LISA conference, Keen et al. [6] described how the High-Performance Computing Center at Michigan State University deployed a centralized research computing platform. Their case study covers many of the important facets of building such a system, including workload characterization, cluster man-

agement and scheduling, network and storage configuration, physical installation, and security. In this report, we look in depth at a particular issue that they touched on only briefly: how to provide a usable software environment to a diverse user base of researchers. We describe the best practices we have used to deploy the software environment on our own cluster, as well as a new system, PyModules, we developed to make this deployment easier. Finally, we speculate on the changes to software management that will occur as more research computing moves from local, university-operated clusters to high-performance computing (HPC) resources that are provisioned in the cloud.

2 Best Practices

As Keen et al. noted, it is common practice to organize the available software on a research compute cluster into modules, with each module representing a specific version of a software package. In fact, this practice dates back nearly 20 years to the Environment Modules tool created by Furlani and Osel [3], which allows administrators to write "modulefiles" that define how a user's environment is modified to access a specific application.

The Environment Modules software makes it possible to install several different versions of the same software package on the same system, allowing users to reliably access a specific version. This is important for stability and for avoiding problems with backward compatibility, especially for software with major changes between versions, such as differences in APIs or the addition or removal of features or default parameters. If only a single version of a software package can be installed at a given time (as is the case for most software package managers for Linux), updating that package to a different version may break users' existing workflows without warning.

While the flexibility introduced by modules is beneficial to both administrators and users, it also creates complexities when modules are used with open-source

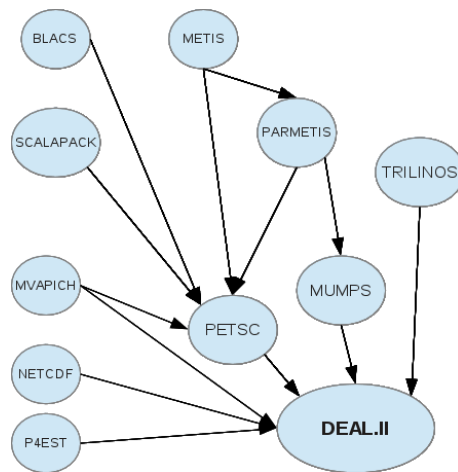


Figure 1: Dependency graph for the deal.II software package.

software.¹ Complexities arise with software configuration and optimization, dependency tracking, and interactions with operating system packages. We address each of these in the subsections below.

2.1 Managing the Configuration and Build Process

Most open-source software installed on CCV’s compute cluster uses one of two configuration and build systems: GNU autoconf² or Kitware’s CMake³ system. Both follow a similar convention that software dependencies are installed in the canonical locations `/usr` or `/usr/local`.

In a research computing software environment with a modules system, this is rarely the case. Instead, dependencies are installed in separate directories in non-canonical locations. Both autoconf and CMake provide mechanisms for specifying the install path of dependencies, but these can lead to very complicated configurations for software with multiple dependencies and sub-dependencies. For example, deal.II [1], a software package installed at CCV and used for analyzing differential equations, has direct or indirect dependencies on roughly ten other software packages (see Figure 1). The dependencies are all installed at different locations in the system. Below is an actual command we used at one point to configure deal.II:

¹Running closed-source software in a modules system is straightforward. Modifying the user’s environment so it contains the path to the executable, and the paths to any license files (if a central licensing server is being used), is usually enough.

²<http://www.gnu.org/software/autoconf/>

³<http://www.cmake.org/>

```

./configure --disable-threads
--with-petsc=/gpfs/runtime/opt/petsc/3.0.0-p12
--with-petsc-arch=linux-gnu-cxx-opt --with-umfpack
--with-trilinos=/gpfs/runtime/opt/trilinos/10.2.2
--with-metis=/gpfs/runtime/opt/metis/4.0.1
--with-blas=goto2 --with-lapack=goto2
--with-p4est=/gpfs/runtime/opt/dealii/7.0.0/p4est
--with-mumps=/gpfs/runtime/opt/mumps/4.9.2
--with-scalapack=/gpfs/runtime/opt/gotoblas2/1.13/lib
--with-blacs=/gpfs/runtime/opt/gotoblas2/1.13/src/BLACS
--enable-mpi CC=mpicc CXX=mpicc
LDFLAGS=-L/gpfs/runtime/opt/gotoblas2/1.13/lib

```

The GNU compilers provide a useful workaround, however, in the form of two environment variables: `CPATH` and `LIBRARY_PATH`. These provide additional directories to search for headers and libraries after the canonical ones, and are also supported by the current versions of other popular compilers, including those from the Intel (2011.11.339) and PGI (12.9) compiler suites. As a result, we specify these variables in any module that contains a library that may serve as a dependency for another library or application. With this setup, complex configuration commands are no longer needed. It is only necessary to have the appropriate dependency modules loaded at compile time.

Setting these environment variables not only makes it easier for administrators to install software packages, but also for users to compile their own custom code. In effect, the variables create an environment that is much closer to the canonical case where all software is installed in `/usr/local` and configuration and building “just works.” However, it retains the added flexibility for upgrading and maintaining multiple versions of software provided by modules.

2.2 Handling Dependencies at Runtime

Software packages that rely on other modules at runtime (for example, dynamically linked libraries) present the same complexity problem at runtime as they do at build time: additional modules must be loaded to satisfy each dependency. This problem can be exacerbated if the software only works with a particular version of a dependency but multiple versions are installed. One possible solution is to include all paths to dependencies in each package’s `LD_LIBRARY_PATH`. However, this leads to modulefiles that quickly grow out of control and present the same readability problems as the deal.II configuration above.

Our preferred solution is to set the `LD_RUN_PATH` variable in the modulefile for any package that provides a library. Then, compiling a dependent package against the library only requires loading the library module at build time. The dependent package will use the library module’s `LD_RUN_PATH` to hard code the absolute path to the library as the default location to search at runtime.

CPU	Highest SSE	# Nodes
Intel Xeon E5540 (Nehalem)	SSE4.2	240
Intel Xeon X5650 (Nehalem)	SSE4.2	34
Intel Xeon X7560 (Nehalem)	SSE4.2	2
Intel Xeon E5-2670 (Sandy Bridge)	AVX	116
AMD Opteron 8382 (Shanghai)	SSE4a	7
AMD Opteron 6282SE (Interlagos)	AVX	2

Table 1: CPU architectures found in CCV’s Oscar compute cluster.

One caveat with using `LD_RUN_PATH` is that moving the library to a different location will break the dependent package. But in a software environment managed by a modules system, the location is typically determined by the package name and version and is rarely moved.

To date, we have successfully used the `LD_RUN_PATH` strategy for all of our library dependencies, even those as complicated as an MPI library (MVAPICH2). The only edge case we have discovered is when a build system passes the `-rpath` flag to the linker. In this case, the `LD_RUN_PATH` value is ignored.

There are two possible solutions. If the hard-coded `-rpath` contains few libraries compared to `LD_RUN_PATH`, the `-rpath` flag can be removed manually and the libraries in it moved to `LD_RUN_PATH`. If `-rpath` contains significantly more libraries, it could be more expedient to add the relevant paths from `LD_RUN_PATH` with additional `-rpath` flags. This is usually as easy as editing a single configuration file, provided the software is using a standard build system. If the software is making use of `-rpath` in a non-standard build system and it is impractical or too difficult to change, then use of `LD_LIBRARY_PATH` should be considered.

2.3 Performance Optimization

On homogeneous clusters, where every node shares the same CPU architecture, software can be compiled using a host-specific optimization flag, such as `-march=native` for the GNU compilers or `-fast` for the Intel compilers. Additionally, many important math libraries have optimized, vendor-provided implementations, such as the AMD Core Math Library⁴ or the Intel Math Kernel Library⁵.

In other cases, however, optimization is not as straightforward. Because hardware procurement can happen in small cycles – especially under a model in which investigators write equipment funding into grants to contribute

to a community cluster – a university research cluster can evolve into a heterogeneous mixture of hardware. At Brown, our research computing platform, Oscar, includes nodes with five similar but distinct CPU architectures, summarized in Table 1. This is in contrast to large, homogeneous systems installed at larger centers like those run by the DOE and NSF.

Conceivably, fully optimizing the software environment to take advantage of the different architectures in a cluster requires localized installations that are specific to each architecture. We have experimented with this at the following three levels of granularity:

- At the coarsest level, using the processor vendor: either Intel or AMD. This allows software to be compiled against the Core Math Library for AMD processors, and the Math Kernel Library for Intel processors.
- At a finer level, using the highest vector instruction set supported by the node. This allows software to take advantage of such instructions if they are available on a node, or otherwise to fall back to a version that works on all nodes.
- At the finest level, using the precise model of processor on the node. This can be used when installing software packages that use autotuning to optimize themselves at build time for a specific processor model.

In practice, though, we have found all of these lacking. Because the Intel Math Kernel Library performs well on all of our AMD and Intel nodes, and automatically selects processor optimizations at runtime, we have abandoned using processor-specific or autotuned alternatives like GotoBLAS [4] and ATLAS [2]. Therefore, neither the first or third levels of localization are necessary.

The second level of localization for the vector instruction set, has not been useful because the most widely used instructions are already available in the older set (SSE3) that is common to all the processors in our cluster.⁶ In one example, we expected several bioinformatics packages that mainly perform string comparisons to benefit from compiling with support for the packed string comparison instructions added in a newer instruction set. What we found instead was that these programs implicitly used the newer instructions through calls to standard C string functions, which are implemented in `glibc` with a mechanism to auto-detect the available instruction set at runtime. Therefore, compiling separate versions for different instruction sets is not necessary.

All of the approaches to performance optimization listed above create a more complicated software environ-

⁴<http://developer.amd.com/tools/cpu-development/amd-core-math-library-acml/>

⁵<http://software.intel.com/en-us/intel-mkl>

⁶For a more detailed comparison of benchmarks across different instruction sets, see <https://bitbucket.org/mhowison/pymodules/src/master/npb-test>.

ment, by requiring multiple versions of a module to support different hardware profiles. Because we have not seen significant gains from them in most cases, we use these optimizations sparingly.

Overall, our optimization strategy has devolved to simply using generic flags like `-O3` and `-msse3` during compilation for most modules.

2.4 Operating System Packages

Much of the software we build depends on libraries provided by operating system packages (from CentOS, in our case). Because the compute nodes in our cluster are diskless, their operating system is installed to a ramdisk. To reduce the footprint of this ramdisk, we install only a minimal set of packages on the compute nodes. We install a fuller set of packages, including development versions of packages, on the login nodes where we compile software. To satisfy runtime dependencies, though, we have to copy many of the libraries provided by the OS packages into a special module, `centos-libs`, that is loaded by default by all users. For most packages, we can use a simple script that parses all of the shared library names from the package's file list and copies them into `centos-libs`. Inevitably, some packages have further complications, for instance because they require additional data in a `share` directory, and we handle these on a case-by-case basis by manually copying additional files into `centos-libs`.

We have only performed one major OS upgrade (from CentOS 5.4 to 6.3), and in this case we chose to rebuild our entire software environment against the newer OS version. While this required substantial effort, it also provided us with two opportunities: (1) to verify how well documented our build processes were, and correct modules that were poorly documented; and (2) to weed out older modules that were no longer in use by our users.

3 PyModules

PyModules is an alternative implementation of the Environment Modules system [3]. We have given PyModules essentially the same syntax and user interface as Environment Modules; however, the backend is written in Python and contains the following improvements:

Simple, INI-style configuration files.

Having created many modulefiles in the Tcl language for the original Environment Modules system, we found they were unnecessarily redundant, since a new file is created per version of an existing package. We designed PyModules to instead use a single INI-style configuration file per package. Multiple versions are defined in

that same file and they can inherit default values, which has simplified our management of these files.

Below is an excerpt from our configuration file for Python, which defines three versions:

```
[DEFAULT]
brief = The Python Programming Language
url = http://www.python.org/
category = languages

prepend PATH = %(rootdir)s/bin
prepend LIBRARY_PATH = %(rootdir)s/lib
prepend LD_LIBRARY_PATH = %(rootdir)s/lib

[2.7.3]
default=true
[3.2.3]
[3.3.0]
```

Our approach gives up some flexibility, since the INI modulefiles cannot execute arbitrary Tcl commands. However, it has the added benefit that we can validate each modulefile before it is available to users. In Environment Modules, it is possible to create modulefiles that generate a Tcl parsing error when loaded.

Improved inventory commands.

Users often need to perform software inventory commands, such as looking up what versions of a specific software package are installed. In PyModules, we cache the parsed modulefiles in an SQLite database, which speeds up both the `module avail` command and the `autocomplete` feature of the `module` command. We also create a `fulltext` index to support wildcard searches on package names and versions. For example, the command `module avail mpi` will show all available modules that start with the token `mpi` in the package name, and the command `module avail mpi/1` will additionally filter only the versions with a 1 in them.

The database is manually updated whenever a new INI configuration file is created or modified, using a new command called `moduledb`. This allows an administrator to review and validate a modulefile before committing it to the live system.

Module categories.

PyModules provides a special field `category` in the INI configuration that can be used to categorize the available modules into bioinformatics packages, physics packages, chemistry packages, etc. The command `module avail` lists all packages broken down by category, and the new command `module avail :category` lists only the packages in the specified category. Modules can belong to multiple categories.

4 Software in the Cloud

Looking forward, cloud-based HPC resources are promising alternatives to university-operated clusters, especially at the scale of small or departmental clusters [5]. Such clusters will require support for applications running on virtualized hardware. This could alleviate many of the problems we have described here, because virtualization provides a new layer of flexibility in customizing the operating system. Instead of providing a single software environment that must span researchers from many disciplines, software environments can be customized by department or research field, and deployed through OS images. Even traditional HPC clusters may in the future provide more support for users to run their own custom OS images.

Shifting the unit of software organization from a software module to an OS image has many implications:

- Administrators will manage a catalog of images for the most common applications. The same mechanisms for versioning and inventory will be needed, but at the OS-image level instead of the module level.
- Versioning of individual software packages may no longer be necessary: instead, an entire OS image can be versioned. This also reduces the complexity of dependencies, since they can be packaged together in the image.
- Since it is no longer necessary to version the software, it could be installed by more canonical means, such as directly to `/usr/local` or using the native OS package manager.
- Tools for automating configuration and builds will become essential, as common packages are re-installed in many OS images.

An important question for building software environments in the cloud is: will the overhead from virtualization degrade application performance? Our experience with the fine-grained optimizations we described in Section 2.3 seems to suggest not: modern software is increasingly using and benefitting from the ability to leverage hardware optimizations, such as vectorization, at runtime. This is corroborated by a study of HPC benchmarks showing that aggressive tuning of the hypervisor can reduce its overhead to only 1% to 5% [7].

Finally, a shift to OS images has important benefits for software distribution. Especially in bioinformatics, there are large-scale challenges to scientific reproducibility caused by incomplete software tools that are difficult to install or use by novice end users. We frequently receive support requests at CCV to install new bioinformatics tools that have complicated dependencies or incomplete build systems. Moving to a paradigm where exper-

imental tools are available via OS images is perhaps the best solution to this problem [8].

5 Conclusion

This report documents the best practices we have arrived at for installing a large collection of scientific applications on our research computing cluster at Brown University. In this context, we have also introduced a new implementation, PyModules, of the Environment Modules system that has helped improve the usability of our software environment. Finally, we have identified possible changes to software management practices resulting from the OS-level virtualization available in the cloud.

Acknowledgments

We thank Paul Krizak for his thorough feedback, which improved the clarity and presentation of the paper.

Availability

PyModules is freely available under a non-commercial license from:

<https://bitbucket.org/mhowison/pymodules>

References

- [1] BANGERTH, W., HARTMANN, R., AND KANSCHAT, G. deal.II - A general-purpose object-oriented finite element library. *ACM Trans. Math. Softw.* 33, 4 (2007).
- [2] CLINT WHALEY, R., PETITET, A., AND DONGARRA, J. J. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing* 27, 1-2 (2001), 3-35.
- [3] FURLANI, J. L., AND OSEL, P. W. Abstract Yourself with Modules. In *Proceedings of the 10th USENIX System Administration Conference (LISA '96)* (Chicago, IL, USA, Sept. 1996).
- [4] GOTO, K., AND GEIJN, R. A. V. D. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.* 34, 3 (2008), 12:1-12:25.
- [5] HILL, Z., AND HUMPHREY, M. A quantitative analysis of high performance computing with Amazon's EC2 infrastructure: The death of the local cluster? In *Proceedings of the 10th IEEE/ACM International Conference on Grid Computing* (Banff, AB, Canada, Oct. 2009), pp. 26-33.
- [6] KEEN, A. R., PUNCH, W. F., AND MASON, G. Lessons Learned When Building a Greenfield High Performance Computing Ecosystem. In *Proceedings of the 26th USENIX Large Installation System Administration Conference (LISA '12)* (San Diego, CA, USA, Dec. 2012).
- [7] KUDRYAVTSEV, A., KOSHELEV, V., PAVLOVIC, B., AND AVETISYAN, A. Virtualizing HPC applications using modern hypervisors. In *Proceedings of the 2012 Workshop on Cloud Services, Federation, and the 8th Open Cirrus Summit (Federated-Clouds '12)* (New York, NY, USA, 2012), ACM, pp. 7-12.
- [8] NOCQ, J., CELTON, M., GENDRON, P., LEMIEUX, S., AND WILHELM, B. T. Harnessing virtual machines to simplify next-generation DNA sequencing analysis. *Bioinformatics* 29, 17 (2013), 2075-2083.