LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-TR-677135

Software Complexity Threatens Performance Portability

T. Gamblin

September 11, 2015

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Software Complexity Threatens Performance Portability

Todd Gamblin, tgamblin@llnl.gov
Lawrence Livermore National Laboratory

Motivation

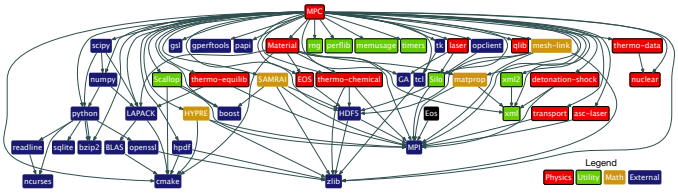
Modern HPC software packages are rarely self-contained. They depend on a large number of external libraries, and many spend large fractions of their runtime in external subroutines. Performance portability depends not only on the effort of application teams, but also on the availability of well-tuned libraries.

At most sites, the burden of maintaining libraries is shared by code teams and facilities. Facilities typically provide well-tuned default versions, but code teams frequently build with bleeding-edge compilers to achieve high performance. For this reason, HPC has no "standard" software stack, unlike other domains where performance is not critical. Incompatibilities among compilers and software versions force application teams and facility staff to re-build custom versions of libraries for each new toolchain. Because the number of potential configurations is combinatorial, and because HPC software is notoriously difficult to port to new machines [3, 7, 8], the tuning effort required to support and maintain performance-portable libraries outstrips the available manpower at most sites. Software complexity is a growing obstacle to performance portability for HPC.

Example Code

Overview. In this section we describe an LLNL-developed multi-physics code, hereafter referred to as *MPC*. MPC is a radiation-hydrodynamics code. Production runs use from 1 to millions of cores. MPC runs on LLNL and LANL machines. These include Intel Linux clusters, Blue Gene/Q, and Cray XE6 systems. It is being ported to LANL's heterogeneous Cray XC30 machine, Trinity.

MPC comprises 47 packages, with complex depedencies (Figure 1a). LLNL-developed packages include 11 physics packages, 4 math/meshing libraries, and 8 utility libraries. MPC also requires 23 open source packages, including MPI, BLAS, and Python. Together, these packages use a diverse set of languages: C, C++, Fortran, Python and Tcl. In all, these packages comprise 2-3 million lines of code.



(a) MPC and its dependency libraries.

	Linux			*BG/Q*	*Cray XE6*
	MVAPICH	*MVAPICH2*	*OpenMPI*	*BG/Q MPI*	*Cray MPI*
GCC	C P L D			C P L D	
Intel 14	C P L D				
Intel 15	C P L D	D			
PGI		D	C P L D		C L D
Clang	C P L D			C L D	
XL				C P L D	

(b) Tested nightly build configurations.

Figure 1: Dependencies and configurations of an LLNL multi-physics code (MPC).

Table 1b shows build configurations that the MPC team tests nightly. The rows and columns show architectures, compilers, and MPI versions. The code builds in four different code configurations, each with slightly different dependencies: current (C) and previous (P) production versions, a "lite" version (L) that includes a smaller set of features, and a development version (D). Some dependencies are needed to fill gaps on particular platforms, such as Blue Gene/Q, which does not provide a sufficiently recent version of Python. In total, 36 different configurations of MPC are built and tested, for a total of around 1,500 different package builds.

Challenges. MPC and other LLNL codes are limited by the manpower required to port, build, and tune software for new systems. Manually building a new configuration of MPC on a new platform has typically required two weeks of effort. This severely limits the number of configurations tested. Manual processes introduce more opportunities for error. Configuration issues, library mismatches, and compiler/MPI incompatibilities dominate porting time.

In the desktop and web computing world, package managers simplify the task of managing large software suites. While there are many OS- and language-specific package managers, tools that handle the complexities of HPC environments are in their infancy. Tools are desperately needed to manage combinatorial, multi-compiler, multi-platform builds. This problem will only grow as new programming models, language features, and runtime systems emerge.

Addressing the Library Portability Challenge

Recently, there has been broader recognition of the software portability problem in HPC, as well as an increased focus on reproducible, reliable deployment of applications. A new set of package management tools has emerged, each attempting to ease the burden of combinatorial HPC software builds [1, 2, 4, 5, 6, 7]. All of these tools encode and archive build "recipes" for different software packages. Once written, the build process can be executed again, so developers do not repeat past mistakes. Many of the tools also encode the dependency relationships and automatically install dependencies with each package. Others parameterize builds, allowing the compiler and build options to be arbitrarily combined. MPC has successfully used the Spack package manager [5] to manage its dependency stack.

Automation. Initially, only Intel/Linux builds of MPC were automated for nightly runs, and builds on other platforms took days or weeks to set up and debug. Only when the team began using Spack to automate builds did running all 36 configurations shown in Table 1b become a practical, routine task.

Reuse. Many of MPC's dependency libraries are shared by other codes at LLNL, and other teams *could* benefit from the MPC team's tuning efforts. However, LLNL libraries are managed by different teams, and some are minimally maintained due to lack of manpower. Contributing patches back to a library's mainline repository was difficult when the library maintainers were not available, or when the library team lacked free cycles to test and verify changes.

Using Spack, the MPC team was able to codify changes to libraries, and to automatically test them without imposing a burden on the library development teams. Each build recipe can be stored in a file and run by others, and developers can easily improve existing recipes without reimplementing an entire build. The cost of porting each library is only incurred once. LLNL teams have created an internal repository of shared builds, further easing the porting burden.

Reproducibility. The performance of large programs depends on how they were built. Tuning builds is a labor-intensive, error-prone process, and part of reproducing the performance of a given run is reproducing its build and runtime environments. Often, this type of provenance information is lost or discarded when performance experiments are run. Automating the build process allow these parameters to be saved for later analysis.

Validation and Testing. After automating their build with Spack, the MPC team was not only able to test more configurations; they were also able to find more bugs. Allowing arbitrary compilers to be swapped into a build enabled the team to easily test with Clang, which was not yet supported by the code. This revealed many incompatibilities in MPC and libraries, well before Clang was actually used as a production compiler.

Conclusions and Recommendations

Using a package management tool to automate the build process has significantly benefitted MPC and other code teams at LLNL. Builds are now reliable, reproducible, and maintainable, and require far less human effort. Teams test more frequently and in more configurations, leading to greater software robustness. Most importantly, LLNL teams now use Spack to share library build recipes, avoiding duplication of effort.

Facility Collaboration. Many packages deployed at LLNL are also deployed at other facilities, and many facilities share similar machines. Currently, efforts to build and tune libraries are largely confined to particular teams and facilities. Tools like Spack provide a vehicle for inter-site sharing and collaboration on performance portability.

The Spack project is open source and collaborative. At LLNL, Spack is being adopted by MPC, other LLNL code teams, tool developers, and facilities staff. We have also seen external interest from other DOE facilities and from other developers on GitHub. We recommend more collaboration among facilities on Spack, Smithy [2], and other similar tools [6, 7]. Consistent deployment methodologies across facilities will aid long-term performance portability.

Tool Development. Spack and other HPC package management tools are in their infancy, and build problems will be even more complex on future machines. HPC package managers need more features and better language, platform, and compiler support for future exascale systems. We recommend funding these projects, as they can save countless hours of HPC developer time. Without funding for HPC package management tools, effort will continue to be duplicated across facilities and code teams, and barriers to portability and performance will remain high.

References

[1] Hashdist. http://github.com/hashdist/hashdist, 2012.

[2] A. DiGirolamo. The Smithy Software Installation Tool. http://github.com/AnthonyDiGirolamo/smithy, 2012.

[3] P. F. Dubois, T. Epperly, and G. Kumfert. Why Johnny Can't Build. *Computing in Science and Engineering*, 5(5):83–88, Sept. 2003.

[4] T. Gamblin. Spack. https://github.com/scalability-llnl/spack, 2014.

[5] T. Gamblin, M. P. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and W. S. Futral. The Spack Package Manager: Bringing order to HPC software chaos. In *Supercomputing 2015 (SC'15)*, Austin, Texas, November 15-20 2015. LLNL-CONF-669890.

[6] M. Geimer, K. Hoste, and R. McLay. Modern Scientific Software Management Using EasyBuild and Lmod. In *Proceedings of the First International Workshop on HPC User Support Tools*, HUST '14, pages 41–51, Piscataway, NJ, USA, 2014. IEEE Press.

[7] K. Hoste, J. Timmerman, A. Georges, and S. De Weirdt. EasyBuild: Building Software with Ease. In *High Performance Computing, Networking, Storage and Analysis, Proceedings*, pages 572–582. IEEE, 2012.

[8] G. Wilson, D. A. Aruliah, C. T. Brown, N. P. C. Hong, M. Davis, R. T. Guy, S. H. D. Haddock, K. Huff, I. Mitchell, M. D. Plumbley, B. Waugh, E. P. White, and P. Wilson. Best Practices for Scientific Computing. *CoRR*, abs/1210.0530, 2012.