Tool Support for Code Evolution and Maintenance of Scientific Software

Peter Pirkelbauer (pirkelbauer2@llnl.gov), Lawrence Livermore National Laboratory
Reed Milewicz (rmilewi@sandia.gov), Sandia National Laboratories

Maintaining and evolving scientific software with respect to new concerns is extremely challenging. Relying purely on manual labor is expensive and error prone. Transformation tools are an important element of the infrastructure requirement of software maintenance. We argue that better practical transformation tools and community platforms on the web will be needed to make high-performance software evolution and maintenance more cost effective.

Introduction

Ensuring the long-term sustainability and stewardship of the DOE scientific software ecosystem means confronting the material realities involved in working with large-scale scientific codes. A state-of-the-art HPC application can consist of millions of lines of code that depend upon an equally vast ecosystem of dependencies to provide capabilities like meshing, discretization, I/O, mathematics, uncertainty quantification, optimization, and visualization. As we saw in the course of the Exascale Computing Project (ECP), evolving and maintaining those codes can be a monumental undertaking. Current advances in tooling and automation notwithstanding, the practice of HPC scientific software development still involves significant manual effort to extend, debug, and optimize codebases -- a costly process of trial and error. Given the unique workforce requirements of computational science, we are limited in our ability to scale up the number of FTEs; as a result, there is an urgent need for new and better approaches to increase productivity in the development and use of HPC applications.

For that reason, we argue that a worthwhile goal of an ASCR software stewardship program would be to develop and promote shared engineering resources to assist with managing large-scale codebases, particularly tooling to enable the transformation and analysis of source code. Source code is the primary instrument and outcome of ASCR investment, and there are ample opportunities for productivity gains in the implementation, debugging, and porting of HPC applications. Doing so would free up teams to focus more on scientific results, mitigate risks due to software defects, and minimize the effort needed to evolve codebases to keep pace with the science. This could include supporting tool development, working with the tooling community to build a consensus around standards and goals, and to train the HPC community on the use of such tools.

This is a risk area for the HPC software ecosystem. Existing commercial, off-the-shelf tools for code transformation and analysis often have little support for the programming languages of interest (e.g., C, C++, and Fortran), struggle to handle the scale and complexity of HPC codebases, or are simply incapable of performing the kinds of context-sensitive transformations and analyses required (e.g., verifying the correctness of MPI applications). On the other hand, while there is a mature and vibrant research community around source-to-source compiler infrastructures for HPC, tool developers often face difficulties in crossing the gap to productionization. While frameworks like ROSE at LLNL enable tool developers to craft solutions for specific applications and use cases, they may not be sufficiently generalizable, reusable, and robust to meet the needs of the broader community (see Milewicz et al. 2021). Meanwhile, in the absence of readily available transformation and analysis tools, HPC researchers and practitioners must expend orders of magnitude more time and effort to accomplish certain tasks. The price of not having the right tools is difficult to quantify, but work by Milewicz and Raybourn provides an illustrative example of how a single MPI bug in a production HPC software stack can take months to find and fix (Milewicz and Raybourn 2018).

Tool Support

Tool support in the form of code transformation tools are very valuable for source code maintenance tasks. We distinguish between transformation tools and analysis tools. Both kinds of tools read source code, transformation tools produce source code as output, while analysis tools report results in terms of source code locations. Transformation tools that produce maintainable source code have the following advantages: (1) the tool can be integrated in any software engineering workflow; the produced code can be checked into a repository and further processed. (2) source code can serve as the common denominator for coupling multiple tools, each specialized for a specific task. (3) The output can be more readily checked and verified by humans. (4) The high-level code remains compiler and runtime system independent.

We note that there are other kinds of tooling, such as those for binary transformation and analysis, which are also of great value to the HPC community and should similarly be promoted. However, source-based automated tooling represents an area of untapped potential. There is a vibrant research community around frameworks and tools for working with source code in HPC contexts, and the same technologies enjoy widespread use in the conventional software industry. Coordinated efforts to advance such tools for use by HPC developers could have a multiplicative effect on productivity. In the software development and maintenance life cycle, such tools can be applied to a variety of tasks; we describe several important use cases.

Code optimizations: ECP efforts have spurred the development and adoption of parallel libraries, such as Raja and Kokkos, that allow users to express node-level parallelism in a portable way. However, architecture specific transformations may still be required to optimize memory access performance for code deployments on a specific system. Performing this step manually would be time consuming and can lead to a proliferation of architecture specific versions - a nightmare for code maintenance. Source-based tooling is well-positioned to work with parallel programming libraries and other constructs in an intelligent way to maximize their use and effectiveness. This includes deep library optimization, library API analysis and instrumentation, library composition, and automated conversion of applications to use different constructs (e.g., OpenMP for node parallelism, CUDA for GPUs). These are all situations where HPC developers too often end up working manually and/or in a piecemeal fashion; supporting the development of capable tools would benefit developers looking to maximize the performance of their code.

Correctness checking of software: Code maintenance poses significant threats to the safety and integrity of software. Even small oversights can lead to bugs and vulnerabilities. The use of third-party libraries that evolve over time poses similar or even greater risks. The often complex interdependencies between software components make the manual development of good test cases hard and testing almost always remains incomplete. Source-based tool support for static analysis (e.g., to exercise code that has been changed compared to earlier versions), dynamic analysis, formal modeling techniques (e.g., model checking, symbolic execution), and full coverage testing would help alleviate concerns. While such tools already exist today, more effort is needed to integrate them in scientific software engineering workflows. User engagement is critical to the adoption of these technologies, to understand how the tools will be used in practice and to engineer the tools for use cases. Strategic investment in productizing and promoting these tools would facilitate that process and accelerate their adoption.

Software adaptation: High-performance computing architectures continuously evolve. To take advantage of new hardware capabilities, existing parallel programming models, for example the Message Passing Interface, are being revised and extended. In turn, software needs to undergo a similar adaptation to take advantage of new library capabilities. However, adapting software applications manually will be prohibitively difficult. Utilizing new library capabilities requires skilled application programmers. Thus, it is essential to produce software development tools and reusable transformations that can be applied to increasingly complex codes. Reusability is key here, as it is relatively easy to define a narrowly-defined mapping between two sets of programming language patterns and constructs and another, but this makes for brittle tools that must be repeatedly redesigned and rebuilt. However, an intentional effort to pool resources into building tools for use by the wider community would result in robust, extensible, and semantics-aware toolkits that could stay current with continual hardware and software evolution.

Dissemination and Training

First, one large obstacle to the adoption of software maintenance tools in HPC is to navigate the plethora of available tools, often research projects, and the correct setup and operation of such tools. One way to mitigate that is through the creation of an online community. Such online communities already exist in some form for other areas of software engineering, for example for software assurance tools (https://continuousassurance.org/). To support potential users of HPC maintenance tools and to connect them with tool developers, an open online community is needed. Such a community would provide a platform where users and developers can interact with each other and exchange ideas. The platform may also host tutorials, in the form of interactive content or videos.

Given the intimate connection between tool developers, tool users, and the problems they aim to solve, we see an opportunity to break down barriers by promoting co-design efforts. As we mentioned previously, user engagement is critical to tool adoption, and efforts to promote meaningful dialogue between stakeholders would be of great benefit. ASCR could spur this dialogue by identifying common challenges of interest to DOE in working with source code, and then pair tool teams with application teams to solve those challenges.

Conclusion

Currently, too much effort is spent on manual software maintenance tasks. Many of these tasks could be simplified through automatic or semi-automatic translators. To avoid a productivity crisis in HPC software, there is an urgent need for better and easy to use tools. Also, building an active user community around software maintenance tools will help to democratize the use of software maintenance tools and achieve the vision of higher software productivity and better HPC software products.

Works Cited

Milewicz, Reed, Peter Pirkelbauer, Prema Soundararajan, Hadia Ahmed, and Tony Skjellum. "Negative Perceptions About the Applicability of Source-to-Source Compilers in HPC: A Literature Review." In *International Conference on High Performance Computing*, pp. 233-246. Springer, Cham, 2021.

Milewicz, Reed, and Elaine Raybourn. "Talk to me: A case study on coordinating expertise in large-scale scientific software projects." *2018 IEEE 14th International Conference on e-Science (e-Science)*. IEEE, 2018.

Acknowledgement

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LLNL-MI-830014.

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 Energy’s National Nuclear Security Administration under contract DENA0003525.