Understanding The High Performance Computing Community: A Software Engineer's Perspective

Victor R. Basili, Jeffrey Carver, Daniela Cruzes, Lorin Hochstein, Jeffrey K. Hollingsworth, Forrest Shull, and Marvin V. Zelkowitz

Introduction

For the last few years we have had the opportunity, as software engineers, to observe development of computational science software (referred to as *codes*) built for High-Performance Computing (HPC) machines in many different contexts. While we do not claim to have studied all types of HPC development, we did encounter a wide cross-section of projects. Despite their diversity, several common traits exist.

1. Many developers receive their software training from other scientists. While the scientists have often been writing software for many years, they generally lack *formal* software engineering training, especially in managing multi-person development teams and complex software artifacts.
2. Many of the codes are not originally designed to be large. They start small and then grow based on their scientific success.
3. A substantial fraction of development teams use their own code (or code developed as part of their research group).

For these reasons (and many others), development practices in this community are quite different from those in more "traditional" software engineering.

Our aim in this paper is to distill our experience about how software engineers can productively engage the HPC community. Several software engineering practices generally considered good ideas in other development environments are quite mismatched to the needs of the HPC community. We found that keys to successful interactions include a healthy sense of humility on the part of software engineering researchers and the avoidance of assumptions that software engineering expertise applies equally in all contexts. We engaged with only a tiny fraction of the HPC community and witnessed enormous variation within and across stakeholders, so generalizations about this "HPC community" must be tempered by understanding just what portion of the community was observed.

Background

The focus of our observations was on computational science codes developed for HPC systems. A list of the 500 fastest (http://www.top500.org) shows that, as of November 2007, the most powerful system has 212,992 processors. While a given application would not routinely use all these processors, it would regularly use a large percentage of them for a single job. Effectively using tens of thousands of processors on a single project is considered normal in this community.

We were interested in codes that require non-trivial communication among the individual processors throughout the execution. While there are many uses for HPC systems, a common application is to simulate physical phenomena, such as earthquakes, global climate change, or nuclear reactions. These codes must be written to explicitly harness the parallelism of HPC systems. While many parallel programming models exist, the dominant model is MPI, a message-passing library where the programmer explicitly specifies all communication. FORTRAN remains widely used for developing new HPC software, as does C/C++. It is not uncommon for a single system to incorporate multiple programming languages, and we even saw several projects use dynamic languages such as Python to couple different modules written in a mix of FORTRAN, C, and C++.

In 2004, DARPA launched the High Productivity Computing Systems (HPCS) project[1] to significantly advance the state of HPC technology by supporting vendor efforts to develop next generation systems, focusing on both hardware and software issues. In addition, DARPA also funded researchers to develop methods for evaluating productivity that more truly measured scientific output rather than simple processor utilization, the measure used by the Top500 list. Our initial role was to evaluate the impact of newly proposed languages on programmer productivity. In addition, one of the authors was involved in conducting a series of case studies of existing HPC projects in government labs to characterize these projects and document lessons learned.

The significance of the HPCS program was its shift in emphasis from *execution time* to *time-to-solution*, which incorporates both development and execution time. We began this research by running controlled experiments to measure the impact of different parallel programming models. Since the proposed languages did not yet exist in a usable form, we performed studies on available technologies such as MPI, OpenMP, UPC, Co-Array FORTRAN, and Matlab*P, using students in parallel programming courses from eight different universities [Ho05].

We widened the scope of this research by collecting "folklore," i.e. the community's tacit, unformalized view about what is true. We collected this folklore first through a focus group of HPC researchers, then by surveying HPC practitioners involved in the HPCS project, and finally by interviewing a sampling of practitioners that included academic researchers, technologists developing new HPC systems, and project managers. Finally, we conducted case studies of projects at both U.S. government labs [Ca07] and academic labs [Ho08].

The development world of the computational scientist

To understand why certain software engineering technologies are a poor fit for computational scientists, it is important to first understand their world and the constraints it places on them. Overall we found that there is no such thing as a single "HPC community." Our research was restricted entirely to computational scientists using HPC systems to run simulations. Yet, despite this narrow focus, we saw enormous variation,

[1] http://www.highproductivity.org

especially in the kinds of problems that people are using HPC systems to solve. Table 1 shows four of the many attributes that vary across the HPC community.

Table 1 – HPC Community Attributes

Attribute	Values	Description
Team Size	Individual	One developer, sometimes called the “lone researcher” scenario
	Large	“Community codes”, multiple groups, possibly geographically distributed
Code Life	Short	Code that is executed few times (e.g. a code from the intelligence community) may trade-off less time in development (spending less time on performance and portability) for more time in execution
	Long	Code that is executed many times (e.g. a physics simulation) will likely spend more time in development (to increase portability and performance) and amortize that time over many executions
Users	Internal	Used only by developers
	External	Used by other groups within the organization (e.g. at U.S. Government Labs) or sold commercially (e.g. Gaussian)
	Both	“Community codes” are used both internally and externally. Version control is more complex in this case because both a development and a release version must be maintained

The goal of scientists is to do science, not execute software

“One possible measure of productivity is scientifically useful results over calendar time. This implies sufficient simulated time and resolution, plus sufficient accuracy of the physical models and algorithms.” –Scientist (interview).

“[Floating-point operations per second] rates are not a useful measure of science achieved.” – User talk, IBM scientific users group conference, as reported in Arctic Research Supercomputing Center, HPC Users Newsletter, 366.

Initially, we believed that performance was of paramount importance to scientists developing on HPC systems. However, after in-depth interviews, we found that scientific researchers are focused on producing publishable results: writing codes that perform efficiently on HPC systems is a means to an end, not an end to itself. While this point may sound obvious, we feel that this is overlooked by many in the HPC community.

A scientist’s goal is to produce new scientific knowledge. Therefore if they can execute their computational simulation using the time and resources they are allocated on the

HPC system, they see no need or benefit in spending time optimizing the performance. The need for optimization is only seen when the simulation cannot be completed at the desired fidelity with the allocated resources. When optimization is necessary, it is often broad-based, including not only traditional computer science notions of code tuning and algorithm modification, but also re-thinking the underlying mathematical approximations and potentially making fundamental changes to the computation. Thus technologies that focus only on code tuning are of somewhat limited utility to this community.

Computational scientists do not view performance gains in the same way as computer scientists. For example, one of the authors (trained in computer science) improved the performance of a code by more than a factor of two. He expected this improvement would save computing time. Instead, when he informed the computational scientist, the reaction was that they could now use the saved time to add more function, i.e., get a higher fidelity approximation of the problem being solved.

Comment: Scientists make decisions based on maximizing scientific output, not program performance.

Performance vs. Portability and Maintainability

If somebody said, maybe you could get 20% [performance improvement] out of it, but you have to do quite a bit of a rewrite, and you have to do it in such a way that it becomes really ugly and unreadable, then maintainability becomes a real problem…. I don't think we would ever do anything for 20%. The number would have to be between 2x and an order of magnitude… Readability is critical in these codes: describe the algorithms in a mathematical language as opposed to a computer language. – Scientist (interview).

Scientists have to balance performance and development effort. We saw preference for technologies that allowed the scientist to control the performance to the level needed for their science, even by sacrificing abstraction and ease of programming. Hence the extensive use of C and FORTRAN, which offer more predictable performance and less abstraction than higher-level programming languages.

Conversely, the scientists are not driven entirely by performance. They will not make significant maintainability sacrifices to obtain modest performance improvements. Because the codes have to run on multiple current and future HPC systems, portability is a major concern. Codes need to run efficiently on multiple machines. Application scientists are not interested in performing machine-specific performance tuning because this effort will be lost when ported to the next platform. In addition, source code changes that improve performance typically make code more difficult to understand, creating a disincentive to do certain kinds of performance improvements.

Comment: Scientists want the control to increase performance as necessary, but will not sacrifice everything to performance.

Verification and validation for scientific codes

Testing is different… it's very much a qualitative judgment about how an algorithm is actually performing in a mathematical sense…. Finally, when the thing is working in a satisfactory way, say, in a single component, you may then go and run it in a coupled application and you'll find out there are some features you didn't understand that came about in a coupled application and you need to go back and think about those. – Scientist (interview).

Simulation software commonly produces an approximation to a set of equations that cannot be solved exactly. One can think of this development as a two-step process: translating the problem to an algorithm and translating the algorithm to code. These approximations (mapping problem to algorithm) can be evaluated qualitatively based on possessing desirable properties (e.g., stability) and ensuring that various conservation laws hold (e.g., that energy is conserved). The required precision of the approximation depends on the nature of the phenomenon being simulated. For example, new problems can arise when approximations of different aspects of a system are integrated. Suddenly, an approximation that was perfectly adequate for standalone usage may not be of sufficient quality for the integrated simulation. Identifying and evaluating the quality of an algorithm is a very challenging task. One scientist we spoke with said that algorithmic defects were much more significant than coding defects.

Validating simulation codes is an enormous challenge. In principle, a code can be validated by comparing the simulation output with the results of a physical experiment. In practice, since simulations are written for domains where experiments are prohibitively expensive or impossible, validation is very difficult. Entire scientific programs, costing hundreds of millions of dollars per year for many years, have been built around experimental validation of large codes.

Comment: *Debugging* and *validation* are qualitatively different for HPC than for traditional software development.

Skepticism of new technologies

I hate MPI, I hate C++. If I had to choose again, I would probably choose the same – Scientist (interview).

Our codes are much larger and more complex than the "toy" programs normally used in [classroom settings]. We would like to see a number of large workhorse applications converted and benchmarked. – Scientist (interview).

Scientists had a cynical view of new technologies because the history of HPC is littered with new technologies that promised to increase scientific productivity but are no longer available. Some of this skepticism is due to the long life of HPC codes; it is not unusual for a code to have a 30 year life cycle. Because of this long life cycle, scientists will only embrace a new technology if they believe it will survive for the long term. This explains the widespread popularity of MPI, despite constant grumbling about its difficulty.

A common strategy used by scientific programmers is to develop code in such a way that different technologies can be plugged in to be tested. For example, when MPI was new in the 1990s, many groups were cautious about its long-term prospects and added it to their code alongside existing message passing libraries. As MPI became widely used and trusted, these older libraries were retired. Similar patterns have been observed with solver libraries, I/O libraries, and tracing tools.

The new languages being developed in the DARPA HPCS projects were intended to extend the frontiers of what is currently possible in today's machines, and so we sought out practitioners working on very large codes and running on very large machines. Because of the time they have already invested in their codes and their need for longevity, they all expressed great trepidation at the prospect of porting to a new language.

Comment: A new technology that can co-exist with older ones has a greater chance of success than one that requires a complete buy-in at the beginning.

Shared, centralized computing resources

"The problem with debugging, of course, is that you want to re-run and re-run. The whole concept of a batch queue would make that a week-long process. Whereas on a dedicated weekend, in a matter of hours you can pound out ten or twenty different runs of enormous size and understand where the logic is going wrong." – Scientist (interview)

Because of the cost, complexity and size of HPC systems, they are typically located at HPC centers and shared among user groups, with batch-scheduling to coordinate executions. Users submit their jobs to a queue with a request for number of processors and maximum execution time. This information is used to determine when to schedule the job. If the time estimate is too low, the job will be preemptively terminated; if it is too high, the job will wait in the queue longer than necessary.

Since these systems are shared resources, scientists are physically remote from the computers they use. Thus, potentially useful tools that were designed to be interactive become unusably slow and are soon discarded, because they are not designed to take into account the long latency times of remote connections. Unfortunately for scientists, using an HPC system typically means interacting with the batch queue.

Debugging batch-scheduled jobs is also tedious because the queue wait increases the turnaround time. Some systems provide "interactive" nodes that allow users to run smaller jobs without entering the batch queue. Unfortunately, some defects only manifest themselves when running with large numbers of processors.

The problem of the queue may be exacerbated by center policies which use *system utilization* as a productivity metric, since utilization is inversely proportional to availability, policies that favor maximizing utilization will have longer waits [Sn06]. As a counterexample, Lincoln Laboratories provides interactive access to all users, and purchases excess computing capacity to ensure that the computational needs of the users are met [Ca07].

Comment: Remote access precludes the use of certain software tools, and system access policies can have a significant impact on productivity.

Computational Science / Software Engineering Mismatches

Repeatedly, we saw that software engineering technologies that did not take into account the constraints of the scientists failed or were not adopted. Our concern is that the computer science community is not necessarily aware of this lesson. Software engineers collaborating with scientists should understand that the resistance to adoption of unfamiliar technologies is based on real experiences. For example, concepts like CMMI are not well matched to the incremental nature of the HPC development process.

Object-oriented languages

Java is for drinking. – Syllabus of a parallel programming course.

Developers on a project said "we're going to use class library X that will hide all our array operations and do all the right things"... Immediately, you ran into all sorts of issues. First of all, C++, for example, was not transportable because compilers work in different ways across these machines. –Scientist (interview).

Object-oriented technologies are firmly entrenched in the software engineering community. But in the HPC community, C and FORTRAN still dominate, although C++ is used and one project was exploring the use of Java. We also saw some use of Python, although never for performance-critical code.

Historically, we see that FORTRAN-like MATLAB has seen widespread adoption among scientists, although not necessarily in the HPC community. To date, OO has not been a good fit for HPC, even though some concepts have been adopted. One possibility may be that OO-based languages such as C++ have been evolving much more rapidly compared to C and FORTRAN in recent years, and are therefore more risky choices.

Comment: More study is needed to identify why OO has seen such little adoption and whether there are pockets within HPC where OO may be suitable

Frameworks

If you talk about components in the Common Component Architecture or anywhere else, components make very myopic decisions. In order to achieve capability, you need to make global decisions. If you allow the components to make local decisions, performance isn't as good. – Scientist (interview)

Frameworks provide a higher-level of abstraction to the programmer, but at a cost of adopting the framework's perspective on how the code should be structured. Examples of HPC frameworks include POOMA (for hiding low-level details of parallelism) and the Common Component Architecture (CCA, for implementing component-based HPC software).

Kendall and Post [Po03] tell the story that Los Alamos National Laboratory sought to modernize an old FORTRAN-based HPC code using POOMA, a novel C++ object-oriented framework for writing parallel codes. Despite spending over 50% of its code development resources during the project on POOMA, the framework was both slower than the original FORTRAN code and lacked the flexibility of the lower-level parallel libraries to implement the desired physics.

In our own studies, we did not encounter scientists using such frameworks. Instead, we saw them implement their own abstraction levels on top of MPI to hide low-level details, and develop their own component architecture to couple their subsystems together.

Of all the multi-physics applications we encountered, only one was using any aspect of CCA technology, and on that project one of the developers was an active member of the CCA initiative. When we asked scientists about the lack of reuse of frameworks such as POOMA, they responded that such frameworks force the scientists to adapt their problem to the interface supported by the framework. They felt that it would take more effort to fit their problem into one of these frameworks than to build their own framework atop lower-level abstraction such as MPI.

One significant barrier to the use of many frameworks is that they cannot be integrated incrementally. As noted earlier, a common risk mitigation strategy is to allow competing technologies to co-exist with a code while under evaluation. However, the nature of many frameworks makes this impossible.

Comment: Scientists have yet to be convinced that reusing existing frameworks will save them more effort than building their own from scratch.

Integrated Development Environments

IDEs try to impose a particular style of development on me and I am forced into a particular mode – Scientist from a U.S. government laboratory [CA07b]

We saw no use of integrated development environments (IDEs) such as Eclipse because they do not fit well into the typical workflow of a scientist running a code on an HPC system. For example, IDEs have no facilities for submitting jobs to remote HPC queues.

IDEs also do not have debugging and profiling support for parallel machines. There is currently an effort to provide this functionality in Eclipse through the Parallel Tools Platform (PTP) project.

In addition, while HPC languages such as FORTRAN and C/C++ are supported in Eclipse, they are second-class citizens in the Eclipse ecosystem which is focused on Java-related technologies. It is an open question whether these technologies will be adopted by the larger HPC system community.

Comment: Without support for remote execution on batch-queued systems, IDEs are unlikely to be adopted by HPC practitioners.

Well-matched technologies are adopted

"We're astrophysicists, which seems to mean we disdain good software engineering practices until we get bit ... hard ... >10 times. Nevertheless, we are starting to learn the importance of source control, regression testing, code verification, and more." - Bronson Messer, "Petascale Supernova Simulation", PETALS workshop, 2006.

"We were using CVS until a few months ago. Now we migrated to Subversion. We've had version control since day 1." – Scientist (interview).

FlashTest, the tool for nightly regression testing of FLASH has been generalized to be usable with any code that uses steps similar to FLASH in building. – ASC/Alliances Center for Astrophysical Thermonuclear Flashes at the University of Chicago: Year 9 Activities Report.

Roccom is an innovative object-oriented, data-centric integration framework developed at CSAR for large-scale numerical scientific simulation. – ASC/Alliances Center for Simulation of Advanced Rockets: 2004 Annual Report.

Scientists do embrace some software engineering techniques and concepts, when they are a good fit. Every multi-developer project we encountered used a version control system such as CVS or Subversion to coordinate changes. We also saw some use of regression testing methods, including tests across platforms and compilers. We saw extensive reuse-in-the-small, in the form of reusing externally developed libraries such as preconditioners, solvers, adaptive mesh refinement support, and parallel I/O libraries.

On multi-physics applications which involved integration of multiple models maintained by independent groups, we saw a lot of effort devoted to software architecture for integrating these components, including use of object-oriented concepts. In one case, an OO language, C++, was explicitly used. In another case an object-oriented architectural framework was implemented using a non-OO language: FORTRAN 90.

Comment: Scientists working on large projects see the value of an architectural infrastructure but they are more disposed to build their own.

What software engineering can do to help scientists

Since this article is aimed at the software engineering community, we conclude with our perspectives on how we software engineers can best apply our knowledge and talents to assist the computational science community.

Observe, identify, adapt, and disseminate good practices and processes

In our study of existing literature, our software environment is not entirely unique. However, our desire to provide an environment that supports development from the inception of high-risk, high-payoff mathematical software to eventual production quality tools is unusual – The Trilinos Software Lifecycle Model, Willenbring et al., Third International Workshop on Software Engineering for High Performance Computing Applications, 2007.

We're doing a loose version of extreme programming or agile. – Scientist (interview).

As software practitioners, we are familiar with the idea of how software development process affects productivity and quality. We can help in two ways. First of all, we can tailor and transfer existing software engineering practices to this community. We know from observation that some mainstream practices have been successfully adopted by larger projects. It is important to publicize these successes. We believe there are other software engineering practices that could be successfully adapted but have not been, such as inspections. Inspections are important here because of the challenges of verification and validation, but they need to be tailored for this domain. As always, it is important to take into account the environment and constraints of this community to avoid mismatches like the ones we have written about. Secondly, we can help capture and disseminate computational-science-specific practices that have been successfully adopted.

Education

Education and outreach to create code that is parallelized - #1 user priority, TeraGrid User Workshop Final Report, July 2006.

Teaching people to use MPI is not very hard. Teaching people to write MPI effectively so that can get performance out of their code is extremely difficult. That's the difference between a first year grad student, and someone who has been at the center for 4-5 years. – Scientist (interview).

For the professor whose job is to turn out students, the correct metric is how long does it take to take a grad student who just finished, say, their second year of coursework to being a productive researcher in the group. That involves a lot more than just actual run-time on the machine. It involves time picking up the skills to be a successful developer, picking up skills as a designer of parallel algorithms, picking up enough physics to understand the problem he's solving and how parallelism applies to it. – Scientist (interview).

We observed both parallel programming classes and HPC practitioners in action. While students learned the basic principles of HPC development in their courses, they were not properly prepared for the kind of software development they needed to do. Therefore, there was a long learning curve to becoming productive, using the apprenticeship model of working closely with more experienced practitioners.

At the university level, we can develop software engineering courses that are specifically targeted at computational scientists. We can also work toward other models of disseminating software engineering knowledge in this domain. For example, we have developed teaching materials that improve the quality of student assignments by teaching them about HPC defects (http://www.hpcbugbase.org),

Research into reuse-in-the-large

A lot of our project is getting all this infrastructure put together that we didn't have [before] and doing this from the ground up. A productive thing would be not to have to do that. - Scientist (interview).

While this paper may have painted a gloomy picture about the prospects of large-scale reuse using frameworks, we believe these technologies have the potential to reduce programmer effort significantly. If a framework is built upon well-supported technologies such as MPI, it will have fewer adoption barriers than a new language.

As software engineers, we can run case studies of projects that attempt to adopt such frameworks. By documenting how and why the adoptions succeed or fail, we can better understand the important context variables for successful framework reuse.

A final word about scope

It is important to note that the scope of observations and conclusions reached in this article are limited to the populations that we interacted with. In particular, we spoke mainly to computational scientists in either academia or government agencies who use computers to do simulations of physical phenomena. There are many other applications for high-performance computing (e.g. signal processing, cryptography, 3D rendering), and HPC is also used in industry (e.g. movie special effects, automobile manufacturers, oil companies).

Scientific software systems are growing larger and more complex. We are finally starting to see interaction among the computational science and software engineering communities, but more dialogue is needed and more studies must be done. We have much to learn from each other.

Acknowledgements

We wish to thank the students who were subjects in the parallel computing classes we studied (Iowa State, Maryland, Mississippi State, MIT, UC Santa Barbara, UCSD, USC, and University of Hawaii). We also thank the code teams we interviewed, including the DOE ASC-Alliance centers (Cal Tech, Stanford University, University of Chicago, UIUC, and University of Utah). We thank the HPC community including users at SDSC, members of the HPCS project, and others for their contributions to our understanding of HPC folklore. We thank DARPA and the US Department of Energy for funding this research. Finally, we thank Bill Dorland for his feedback on an earlier draft of this paper.

References

[Ca07] J. Carver, "Post-workshop report for the third international workshop on software engineering for high performance computing applications (SE-HPC 07)," SIGSOFT Softw. Eng. Notes, vol. 32, pp. 38-43, 2007.

[Ca07b] J. Carver, R. Kendall, S. Squires, and D. Post, “Software Development Environments for Scientific and Engineering Software: A Series of Case Studies,” Proceedings of the 29th International Conference on Software Engineering. p. 550-559. 2007.

[Ho05] Lorin Hochstein, Jeff Carver, Forrest Shull, Sima Asgari, Victor R. Basili, Jeffrey K. Hollingsworth, Marvin Zelkowitz. "Parallel Programmer Productivity: A Case Study of Novice Parallel Programmers". International Conference for High Performance Computing, Networking and Storage (SC'05). November 2005.

[Ho08] L. Hochstein, V. R. Basili, “The ASC-Alliance projects: A case study of large-scale parallel scientific code development”, IEEE Computer, March 2008.

[Po04] D. E. Post and R. P. Kendall, "Software project management and quality engineering practices for complex, coupled multiphysics, massively parallel computation simulations: Lessons learned from ASCI," International Journal of High Performance Computing Applications, vol. 18, pp. 399-416, 2004.

[Sn06] A. Snavely and J. Kepner, "Is 99% utilization of a supercomputer a good thing?," in Proceedings of the 2006 ACM/IEEE conference on Supercomputing Tampa, Florida: ACM, 2006.