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

Victor R. Basili and Daniela Cruzes, *University of Maryland, College Park, and Fraunhofer Center for Experimental Software Engineering-Maryland*

Jeffrey C. Carver, *Mississippi State University*

Lorin M. Hochstein, *University of Nebraska-Lincoln*

Jeffrey K. Hollingsworth and Marvin V. Zelkowitz, *University of Maryland, College Park*

Forrest Shull, *Fraunhofer Center for Experimental Software Engineering-Maryland*

Computational scientists developing software for HPC systems face unique software engineering issues. Attempts to transfer SE technologies to this domain must take these issues into account.

For the past few years, we've had the opportunity, as software engineers, to observe the development of computational-science software (called *codes*) built for high-performance-computing (HPC) machines in many different contexts. Although we haven't studied all types of HPC development, we've encountered a wide cross-section of projects. Despite these projects' diversity, several common traits exist:

- Many developers receive their software training from other scientists. Although the scientists often have been writing software for many years, they generally lack formal software engineering (SE) training, especially in managing multiperson development teams and complex software artifacts.
- Many of the codes aren't initially designed to be large. They start small and then grow on the basis of their scientific success.
- Many 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 differ considerably from those in more "traditional" SE.

We aim here to distill our experience about how software engineers can productively engage the HPC community. Several SE practices generally considered good ideas in other development environments are quite mismatched to the HPC community's needs. For SE researchers, the keys to successful interactions include a healthy sense of humility and the avoidance of assumptions that SE expertise applies equally in all contexts.

Background

A list of the 500 fastest supercomputers (www.top500.org) shows that, as of November 2007, the

0740-7459/08/$25.00 © 2008 IEEE

Table 1

HPC community attributes

Attribute	Values	Description
Team size	Individual	This scenario, sometimes called the "lone researcher" scenario, involves only one developer.
	Large	This scenario involves "community codes" with multiple groups, possibly geographically distributed.
Code life	Short	A code that's executed few times (for example, one from the intelligence community) might trade less development time (less time spent on performance and portability) for more execution time.
	Long	A code that's executed many times (for example, a physics simulation) will likely spend more time in development (to increase portability and performance) and amortize that time over many executions.
Users	Internal	Only developers use the code.
	External	The code is used by other groups in the organization (for example, at US government labs) or sold commercially (for example, Gaussian, www.gaussian.com)
	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.

most powerful system had 212,992 processors. Although a given application wouldn't routinely use all these processors, it would regularly use a high 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 requiring nontrivial communication among the individual processors throughout the execution. Although HPC systems have many uses, 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 HPC systems' parallelism. Although many parallel-programming models exist, the dominant model is MPI (message-passing interface), a library where the programmer explicitly specifies all communication. Fortran remains widely used for developing new HPC software, as do C and C++. Frequently, a single system incorporates multiple programming languages. 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 program (HPCS, www.highproductivity.org) to significantly advance 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 productivity evaluation methods that measure scientific output more realistically than does simple processor utilization, the measure used by the Top500 list. Our initial role was to evaluate how newly proposed languages affect programmer productivity. In addition, one of us helped conduct a series of case studies of existing HPC projects in government labs to characterize these projects and document lessons learned.

The HPCS program's significance 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. Because the proposed languages weren't yet usable, we studied available technologies such as MPI, OpenMP, UPC (Unified Parallel C), Co-Array Fortran, and Matlab*P, using students in parallel-programming courses from eight different universities.[1]

To widen this research's scope, we collected "folklore"—that is, the community's tacit, unformalized view of what's true. We collected it first through a focus group of HPC researchers, then by surveying HPC practitioners involved in the HPCS program, and then by interviewing a sampling of practitioners including academic researchers, technologists developing new HPC systems, and project managers. Finally, we conducted case studies of projects at both US government labs[2] and academic labs.[3]

The development world of the computational scientist

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

Scientists want the control to increase performance as necessary but won't sacrifice everything to performance.

variation, 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.

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. (All quotes are from interviews with scientists unless otherwise noted.)*
>
> *[Floating-point operations per second] rates are not a useful measure of science achieved.—user talk, IBM scientific users group conference*[4]

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 focus on producing publishable results. Writing codes that perform efficiently on HPC systems is a means to an end, not an end in itself. Although this point might sound obvious, we feel that many in the HPC community overlook it.

A scientist's goal is to produce new scientific knowledge. So, if scientists can execute their computational simulations using the time and resources allocated to them on the HPC system, they see no need for or benefit from optimizing the performance. They see the need for optimization only when they can't complete the simulation at the desired fidelity with the allocated resources. When optimization is necessary, it's often broad-based, not only including traditional computer science notions of code tuning and algorithm modification but also rethinking the underlying mathematical approximations and potentially fundamentally changing the computation. So, technologies that focus only on code tuning are of somewhat limited utility to this community.

Computational scientists don't view performance gains in the same way as computer scientists. For example, one of us (trained in computer science) improved a code's performance by more than a factor of two. He expected this improvement would save computing time. Instead, when he informed the computational scientist, the scientist responded that the saved time could be used to add more function—that is, to get a higher-fidelity approximation of the problem being solved.

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

Performance vs. portability and maintainability

> *If somebody said, maybe you could get 20 percent [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 percent. 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.*

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

Conversely, the scientists aren't driven entirely by performance. They won't sacrifice significant maintainability for modest performance improvements. Because the codes must run on multiple current and future HPC systems, portability is a major concern. Codes must run efficiently on multiple machines. Application scientists aren't interested in performing machine-specific performance tuning because they'll lose the benefits of their efforts when they port the code to the next platform. In addition, source code changes that improve performance typically make code more difficult to understand, creating a disincentive to make certain kinds of performance improvements.

Conclusion: Scientists want the control to increase performance as necessary but won't 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 fea-*

tures you didn't understand that came about in a coupled application and you need to go back and think about those.

Simulation software commonly produces an approximation to a set of equations that can't be solved exactly. You can think of this development as a two-step process: translating the problem to an algorithm and translating the algorithm to code. You can evaluate these approximations (mapping a problem to an algorithm) qualitatively on the basis of possessing desirable properties (for example, stability) and ensuring that various conservation laws hold (for example, that energy is conserved). The approximation's required precision depends on the nature of the phenomenon you're simulating. For example, new problems can arise when you integrate approximations of a system's different aspects. Suddenly, an approximation that was perfectly adequate for standalone use might not be good enough for the integrated simulation. Identifying and evaluating an algorithm's quality is a challenge. One scientist we spoke with said that algorithmic defects are much more significant than coding defects.

Validating simulation codes is an enormous challenge. In principle, you can validate a code by comparing the simulation output with a physical experiment's results. In practice, because simulations are written for domains in which 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.

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

Skepticism of new technologies

I hate MPI, I hate C++. [But] if I had to choose again, I would probably choose the same.

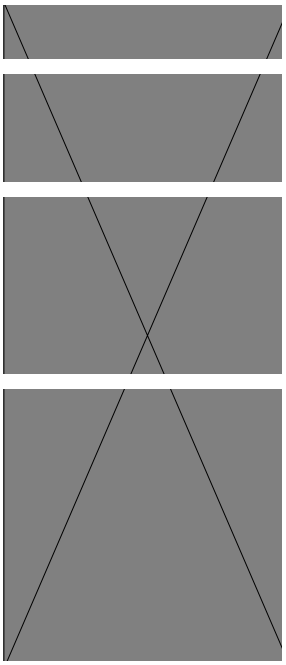
Our codes are much larger and more complex than the "toy" programs normally used in [classroom settings]. We would like to see a number of large workhorse applications converted and benchmarked.

The scientists have a cynical view of new technologies because the history of HPC is littered with new technologies that promised increased scientific productivity but are no longer available. Some of this skepticism is also due to the long life of HPC codes; frequently, a code will have a 30-year life cycle. Because of this long life, scientists will embrace a new technology only if they believe it will survive in the long term. This explains MPI's widespread popularity, despite constant grumbling about its difficulty.

Scientific programmers often develop code such that they can plug in different technologies to evaluate them. 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 languages being developed in the DARPA HPCS program were intended to extend the frontiers of what's possible in today's machines. So, we sought practitioners working on very large codes running on very large machines. Because of the time they've already invested in their codes and their need for long-lived codes, they all expressed great trepidation at the prospect of rewriting a code in a new language.

Conclusion: A new technology that can coexist with older ones has a greater chance of success than one requiring complete buy-in at the beginning.



Shared, centralized computing resources

The problem with debugging, of course, is that you want to rerun and rerun. 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 10 or 20 different runs of enormous size and understand where the logic is going wrong.

Because of HPC systems' cost, complexity, and size, they're 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 a certain 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's too high, the job will wait in the queue longer than necessary.

Because these systems are shared resources, scientists are physically remote from the computers they use. So, potentially useful tools that were designed to be interactive become unusably slow and are soon discarded because they don't take

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

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 let users run smaller jobs without entering the batch queue. Unfortunately, some defects manifest themselves only when the code runs on large numbers of processors.

Center policies that use *system utilization* as a productivity metric might exacerbate the problem of the queue. Because utilization is inversely proportional to availability, policies that favor maximizing utilization will have longer waits.[5] As a counterexample, Lincoln Laboratories provides interactive access to all users and purchases excess computing capacity to ensure that users' computational needs are met.[2]

Conclusion: Remote access precludes the use of certain software tools, and system access policies can significantly affect productivity.

Mismatches between computational science and SE

Repeatedly, we saw that SE technologies that don't take the scientists' constraints into account fail or aren't adopted. The computer science community isn't 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 such as CMMI aren't well matched to the incremental nature of HPC development.

Object-oriented languages

> *Java is for drinking.—parallel-programming course syllabus*

> *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.*

OO technologies are firmly entrenched in the SE 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 Python use, although never for performance-critical code.

Fortran-like Matlab has seen widespread adoption among scientists, although not necessarily in the HPC community. To date, OO hasn't been a good fit for HPC, even though the community has adopted some concepts. One reason for the lack of widespread adoption might be that OO-based languages such as C++ have been evolving much more rapidly than C and Fortran in recent years and are therefore riskier choices.

Conclusion: More study is needed to identify why OO has seen such little adoption and whether pockets exist in HPC where OO might 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.*

Frameworks provide programmers a higher level of abstraction, but at the cost of adopting the framework's perspective on how to structure the code. Example HPC frameworks include

- POOMA (Parallel Object-Oriented Methods and Applications), a novel C++ OO framework for writing parallel codes that hides the low-level details of parallelism, and
- CCA (Common Component Architecture), for implementing component-based HPC software.

Douglass Post and Richard Kendall tell how Los Alamos National Laboratory sought to modernize an old Fortran-based HPC code using POOMA.[6] Even though the project spent over 50 percent of its code-development resources on POOMA, the framework was slower than the original Fortran code. It also lacked the flexibility of the lower-level parallel libraries to implement the desired physics.

The scientist in our studies don't use frameworks. Instead, they implement their own abstraction levels on top of MPI to hide low-level details, and they develop their own component architecture to couple their subsystems.

Of all the multiphysics applications we encountered, only one used any aspect of CCA technology, and one of that application's 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 them to adapt their problem to the interface supported by the framework. They feel that fitting their problem into one of these frameworks will take more effort than building their own framework atop lower-level abstractions such as MPI.

For many frameworks, a significant barrier to their use is that you can't integrate them incrementally. As we noted earlier, a common risk-mitigation strategy is to let competing technologies coexist with a code while they're under evaluation. However, the nature of many frameworks makes this impossible.

Conclusion: 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.—US government laboratory scientist*[7]

We saw no use of integrated development environments (IDEs) such as Eclipse because they don't 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 don't support debugging and profiling for parallel machines. The Eclipse Parallel Tools Platform project (www.eclipse.org/ptp) is attempting to provide this functionality.

In addition, although Eclipse supports HPC languages such as Fortran and C/C++, they're second-class citizens in the Eclipse ecosystem, which focuses on Java-related technologies. Whether the larger HPC-system community will adopt these technologies is an open question.

Conclusion: Unless IDEs support remote execution on batch-queued systems, HPC practitioners won't likely adopt them.

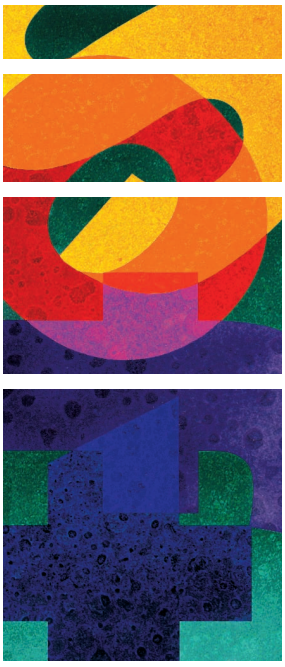
But well-matched technologies are adopted

> *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.*[8]

> *We were using CVS until a few months ago. Now we migrated to Subversion. We've had version control since day 1.*

> *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.*[9]

> *Roccom is an innovative object-oriented, data-centric integration framework developed at CSAR [the Center for Simulation of Advanced Rockets] for large-scale numerical scientific simulation.*[10]



Scientists do embrace some SE techniques and concepts, when they're a good fit. Every multideveloper 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 multiphysics applications involving integration of multiple models maintained by independent groups, the scientists devoted much effort on software architecture for integrating these components, including using OO concepts. In one case, they explicitly used an OO language, C++. In another case, they implemented an OO architectural framework using a non-OO language—Fortran 90.

Conclusion: Scientists working on large projects see the value of an architectural infrastructure, but they're more disposed to build their own.

What SE can do to help scientists

So, how can we software engineers best apply our knowledge and talents to assist the computational-science community?

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.*[11]

Larger projects have successfully adopted some mainstream practices. It's important to publicize these successes.

We're doing a loose version of Extreme Programming or agile.

As software practitioners, we're familiar with how the software development process affects productivity and quality. We can help in two ways.

First, we can tailor and transfer existing SE practices to the HPC community. We know from observation that larger projects have successfully adopted some mainstream practices. It's important to publicize these successes. Other SE practices, such as inspections, could be successfully adapted but haven't been. Inspections are important here because of the challenges of verification and validation, but they must be tailored for this domain. As always, it's important to take into account this community's environment and constraints to avoid mismatches such as the ones we've mentioned.

Second, 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.[12]

Teaching people to use MPI is not very hard. Teaching people to write MPI effectively so that [they] 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 four to five years.

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 be a productive researcher in the group? That involves a lot more than just actual runtime 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.

We observed both parallel-programming classes and HPC practitioners in action. Although students learned the basic principles of HPC development in their courses, they weren't properly prepared for the kind of software development they needed to do. So, there was a long learning curve to becoming productive, involving the apprenticeship model of working closely with more experienced practitioners.

At the university level, we can develop SE courses specifically for computational scientists. We can also work toward other models of disseminating SE knowledge in this domain. For example, to improve the quality of students' assignments, we've developed materials for teaching them about HPC defects (www.hpcbugbase.org).

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.

Although we could have painted a gloomy picture here about the prospects of large-scale reuse of frameworks, we believe these technologies could reduce programmer effort significantly. A framework that's built on well-supported technologies such as MPI 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.

The scope of observations and conclusions we present here are limited to the populations with which we interacted. We spoke mainly to computational scientists in either academia or government agencies who use computers to simulate physical phenomena. There are many other applications for HPC (for example, signal processing, cryptography, and 3D rendering), and HPC is also used in industry (for example, movie special effects, automobile manufacturing, and oil production).

Scientific-software systems are growing larger and more complex. We're finally starting to see interaction among the computational-science and SE communities, but more dialogue and studies are necessary. We've much to learn from each other.

Acknowledgments

We thank the students who were subjects in the parallel-computing classes we studied (Iowa State Univ.; Univ. of Maryland; Mississippi State Univ.; Massachusetts Inst. of Technology; Univ. of California, Santa Barbara; University of California, San Diego; University of Southern California; and University

of Hawaii). We also thank the code teams we interviewed, including the US Department of Energy ASC (Advanced Simulation and Computing) Alliance centers (California Inst. of Technology, Stanford Univ., Univ. of Chicago, Univ. of Illinois Urbana-Champaign, and Univ. of Utah). We thank the high-performance-computing community, including users at SDSC, members of the High Productivity Computing Systems program, 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 article.

References

1. L. Hochstein et al., "Parallel Programmer Productivity: A Case Study of Novice Parallel Programmers," *Proc. Int'l Conf. High Performance Computing, Networking, and Storage* (SC 05), IEEE CS Press, 2005, p. 35.
2. J. Carver, "Post-Workshop Report for the Third International Workshop on Software Engineering for High Performance Computing Applications (SE-HPC 07)," *SIGSOFT Software Eng. Notes*, vol. 32, no. 5, 2007, pp. 38–43.
3. L. Hochstein and V.R. Basili, "The ASC-Alliance Projects: A Case Study of Large-Scale Parallel Scientific Code Development," *Computer*, Mar. 2008, pp. 50–58.
4. T. Logan, "Highlights from SCICOMP-13, the IBM Scientific Users Group," *Arctic Research Supercomputing Center HPC Users Newsletter*, no. 366, 2007, www.arsc.edu/support/news/HPCnews/HPCnews366.shtml.
5. A. Snavely and J. Kepner, "Is 99% Utilization of a Supercomputer a Good Thing?" *Proc. 2006 ACM/IEEE Conf. Supercomputing*, ACM Press, 2006, article 37.
6. 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," *Int'l J. High Performance Computing Applications*, Winter 2004, pp. 399–416.
7. J. Carver et al., "Software Development Environments for Scientific and Engineering Software: A Series of Case Studies," *Proc. 29th Int'l Conf. Software Eng.*, IEEE CS Press, 2007, pp. 550–559.
8. B. Messer, "Petascale Supernova Simulation," presentation at 2006 PETALS (*Peta*scale *L*aboratory for *S*imulation Science) Workshop, 2006, www.jics.utk.edu/petals/Messer.pdf.
9. *Year 9 Activities Report*, ASC/Alliances Center for Astrophysical Thermonuclear Flashes, Univ. of Chicago, 2006, http://flash.uchicago.edu/website/information/AnnualReport06.pdf.
10. *2004 Annual Report*, ASC/Alliances Center for Simulation of Advanced Rockets, Univ. of Illinois, 2004, www.csar.uiuc.edu/annual_reports/AnnReport04/index.html.
11. J.M Willenbring, M.A. Heroux, and R.T. Heaphy, "The Trilinos Software Lifecycle Model," *Proc. 3rd Int'l Workshop Software Eng. for High Performance Computing Applications* (SE-HPC 07), IEEE CS Press, 2007, p. 6.
12. A. Zimmerman and T.A. Finholt, *TeraGrid User Workshop Final Report*, Collaboratory for Research on Electronic Work, School of Information, Univ. of Michigan, 2006, www.crew.umich.edu/research/teragrid_user_workshop.pdf.

About the Authors

Victor R. Basili is a professor of computer science at the University of Maryland and a research fellow at the Fraunhofer Center for Experimental Software Engineering. He works on measuring, evaluating, and improving the software process and product. Basili received his PhD in computer science from the University of Texas at Austin. Contact him at basili@cs.umd.edu.

Jeffrey C. Carver is an assistant professor in Mississippi State University's Department of Computer Science and Engineering. His research interests include empirical software engineering, software engineering for computational science, software architecture, requirements engineering, process improvement, and computer security. Carver received his PhD in computer science from the University of Maryland. He's a member of the IEEE Computer Society, the ACM, and the American Society for Engineering Education. Contact him at carver@cse.msstate.edu.

Daniela Cruzes is a postdoctoral fellow at the University of Maryland and Fraunhofer Center for Experimental Software Engineering, Maryland. Her research interests are empirical software engineering, data mining, systematic reviews, data analysis, and knowledge management. Contact her at dcruzes@fc-md.umd.edu.

Lorin M. Hochstein is an assistant professor in the Department of Computer Science and Engineering at the University of Nebraska at Lincoln, where he's a member of the Laboratory for Empirically Based Software Quality Research and Development. His research interests include empirical software engineering and software engineering for high-performance computing. Hochstein received his PhD in computer science from the University of Maryland. He's a member of the IEEE Computer Society and the ACM. Contact him at lorin@cse.unl.edu.

Jeffrey K. Hollingsworth is a professor in the Computer Science Department at the University of Maryland, College Park, and is affiliated with the University's Department of Electrical Engineering and Institute for Advanced Computer Studies. His research interests include instrumentation and measurement tools, resource-aware computing, high-performance distributed computing, and programmer productivity. Hollingsworth received his PhD in computer science from the University of Wisconsin. He's a senior member of the IEEE and a member of the ACM. Contact him at hollings@umd.edu.

Forrest Shull is a senior scientist at the Fraunhofer Center for Experimental Software Engineering, Maryland, and director of its Measurement and Knowledge Management division. His research interests include software inspections and empirical software engineering. Shull received his PhD in computer science from the University of Maryland. Contact him at fshull@fc-md.umd.edu.

Marvin V. Zelkowitz is a research professor in the University of Maryland's Computer Science Department. His research interests include technology transfer and experimental software engineering. Zelkowitz received his PhD in computer science from Cornell University. He's a member of the ACM and a fellow of the IEEE. Contact him at mvz@cs.umd.edu.