Scientific Computing's Productivity Gridlock: How Software Engineering Can Help

Hardware improvements do little to improve real productivity in scientific programming. Indeed, the dominant barriers to productivity improvement are now in the software processes. To break the gridlock, we must establish a degree of cooperation and collaboration with the software engineering community that does not yet exist.

Large-scale highly parallel scientific program developers have long struggled with a productivity crisis: machines grow bigger and faster, but it gets more and more difficult to get useful work done. The DARPA High Productivity Computing Systems (HPCS) program challenged industry vendors to design a dramatically different kind of petascale computing system. According to HPCS, such a system should be faster in the traditional hardware sense, as well as 10 times more productive at supporting scientific programming applications of strategic national importance.[1] Acknowledging that productivity is poorly understood, DARPA further challenged the vendors to collaborate with the research community to develop an understanding of productivity that could further guide and help evaluate high-end systems design.

1521-9615/09/$26.00 © 2009 IEEE
Copublished by the IEEE CS and the AIP

Stuart Faulk
University of Oregon
Eugene Loh and Michael L. Van De Vanter
Sun Microsystems
Susan Squires
Tactics, LLC
Lawrence G. Votta
Brincos

Sun Microsystems took a broad view of the productivity problem. Guided by software and hardware technology studies, we assembled a team of researchers with expertise in cultural anthropology, physics, scientific programming, empirical software engineering, software development technologies, and programming languages. We then studied the missions, technologies, and practices at government-funded institutions (DARPA's "mission partners"). These institutes represent scientific computing's highest end—that is, they have the biggest machines and the strongest imperative to maximize both hardware utilization and large-scale parallelism.

Our results helped us design a new class of machine for productive high-end scientific programming.[2] More broadly, we gained insight into the productivity problem's nature and underlying causes, as well as what they imply about how we might navigate beyond this crisis. These insights are our focus here.

Although the high-performance computing community typically emphasizes hardware issues, our findings suggest that the dominant barriers to productivity improvement are in the software processes. The development environment's unique goals and constraints have led the scientific programming community to evolve its own characteristic software development approach. As our workflow studies show, this approach creates bottlenecks, imposing critical constraints on developers' ability to improve real (end-to-end) productivity. Moreover, these bottlenecks are inherent in the approach—particularly in its reliance on having multidisciplinary experts handcraft the code—and hence we can't remove the blocks without fundamentally changing the way scientific codes are developed.

Although the software engineering community can help here, doing so will require a level of collaboration and cooperation between the software engineering and scientific computing communities that currently doesn't exist.

Studying Productivity: The Scientific Basis

A primary objective of Sun's approach has been to establish a sound scientific basis for studying software productivity in the high-performance computing domain. DARPA's programmatic goal was to address "real productivity," which it defined as "the ability to develop and deploy high-performance supercomputer applications at an acceptable time and cost."[1] While recognizing that productivity was ill defined, the goal established that the problem's scope was not just hardware performance, but also all aspects of the software development life cycle. Such a scope includes software development's human and organizational issues, the system administration, and the scientists who would use the system.

Given this scope, we identified two important goals:

- embrace the broadest possible view of productivity, including not only customary metrics—such as peak hardware speed and resource utilization—but also human tasks, skills, motivations, organizations, and culture; and
- establish an investigation on the soundest possible scientific basis, drawing on established research methodologies from all the relevant fields, including those unfamiliar within many research communities. (One example here is time and motion studies of software developers at work.[3])

To accomplish these goals, we assembled an interdisciplinary team with broad expertise across the social, physical, and computational sciences. This team created a scientific framework for exploiting multiple research disciplines appropriate to the phenomena under investigation.[4] Our work is thus grounded in empirical data, validated by multiple approaches ("triangulation"), and most importantly, applied to the professionals that the work actually targets.

We've described details of the research paradigm, the studies, and their results elsewhere.[2] Here, we've synthesized our findings to convey a broad understanding of the productivity problem's nature from the intuitive perspective as a process for producing scientific results. Our problem statement and recommendations are thus necessarily broad as well, but are based not on anecdotal evidence but rather represent conclusions drawn from careful study.

Scientific Computing's Productivity Crisis

In studying scientific programming practices, we encountered people possessed of extraordinary skill, dedication, and resourcefulness in their pursuit of strategically important missions that routinely entailed unprecedented programming challenges. Simultaneously, we saw many manifestations of a productivity crisis: frustratingly long and troubled software development times, a growing shortage of expertise in critical areas,[5] a dysfunctional market in supporting tools,[6] acute problems in achieving required portability (both portable software and portable programming skills), and growing concern about the reliability of scientific results based on that software.[7]

Although the broader computing community has experienced and addressed many of these issues in other domains, a "software chasm" has historically inhibited knowledge transfer into the scientific programming domain, where modern software engineering practices scarcely exist.[8] This *communication gap* is a recurring theme throughout our findings; it's a gap grounded in the diverse values and constraints of the scientific and general computing communities (from which scientific programmers evidently seceded decades ago).

So, rather than adapt and apply software engineering technologies (such as processes, methods, and tools), scientific programmers overwhelmingly favor handcrafted solutions because "the computer scientists don't address our needs" and "there isn't enough money." Although such views might be justified by experience, they've isolated the scientific programming community from much needed help. The community has essentially become stuck at local optima—that is, approaches such as platform-specific optimizations that might help address near-term problems but that provide no clear path to (and indeed, often inhibit) global productivity improvement. This inability to start solving productivity problems, even as overall productivity declines, is the so-called *productivity gridlock* in scientific computing.

The Communication Gap

Our research shows two cultures "separated by a common language." Visualize for a moment scientific computing as an isolated island, colonized by explorers from afar and then abandoned for decades. Returning visitors (software engineers) find the inhabitants (scientific programmers) apparently speaking the same language, but communication—and thus collaboration—is nearly impossible; the technologies, culture, and language semantics themselves have evolved and adapted to circumstances unknown to the original colonizers.

The visitors marvel at the islanders' intelligent adaptation and extraordinary accomplishments, but are simultaneously appalled at the primitive conditions in which they live. The islanders, for their part, often laugh at the visitors, whose boatloads of modern technologies will surely fail to help them survive the harsh local environment.

Divergent values and constraints nearly preclude effective communication: one side sees the other as backward and stubborn; the other side sees the former as arrogant and irrelevant. Both are correct, within their own frames of reference, but the status quo offers little possibility for improvement.

Scientific Programming vs. Software Engineering

In visiting the scientific programming community, software engineers note two striking differences from other computing domains: the tools, practices, and even the vernacular are unfamiliar, and few people seem to be trained primarily in computer science.

We gathered a baseline of data about scientific computing environments—including several government-funded laboratories[9]—and identified their distinctive characteristics. As the following list shows, some characteristics are unique to scientific programming, but many differ mainly in relative emphasis.

- *Words mean different things.* Scientific programmers write "codes." A "serial" code doesn't use parallelism. "Scaling" a serial code rewrites and adapts it for parallel execution. Such terms are unknown in the general software engineering community.
- *It's about the science.* Codes are valued only to the extent that they efficiently solve problems at hand.
- *Scientific codes are expensive.* Small teams of two to eight highly trained professionals (not software engineers) might spend four to six years developing new mathematical models, adapting numerical techniques, and creating a highly parallel code to solve a particular problem.
- *Codes are long lived.* A successful scientific code might be maintained, evolved, and extended for 20 to 30 years, as long as it solves problems for the "customers."
- *Performance really matters.* Addressing ever larger and more complex problems, scientific programmers focus almost exclusively on performance. Typically half of a project's development time is spent scaling and optimizing an otherwise sound scientific code. Manual optimizations, machine-specific tricks, and mathematical shortcuts are common, possibly binding the code closely to a particular machine and software environment. The only development tools that matter are dedicated to performance.
- *Hardware platforms change often.* A high-end machine typically becomes obsolete in four years. Codes port often, even during initial development, and might require significant rework to improve performance.
- *Portability also really matters.* In perpetual conflict with performance goals, developers direct great effort toward code portability. The road is littered with once useful codes for which porting ceased to be funded.
- *It's all Fortran 77 and C++.* New projects still commit to these languages and, once committed, don't change. These are the safe choices: the only two likely to be supported on every system for decades into the future.
- *Hardware costs dominate.* Personnel costs play a small role in hardware decisions. The recent migration from shared to distributed memory machines was motivated by inexpensive microprocessors, but came at the expense of more difficult programming. An example here is requiring standardization on a low-level message-passing interface (MPI) library for parallelization.

The properties that scientific programmers consider critical (performance, hardware costs, and portability) are among the least significant to most software engineers. In fact, current software engineering practice intentionally abstracts away from such hardware properties. Conversely, issues important to software engineers (highly maintainable code, robust programming languages and practices, increasingly higher abstraction levels, time to market) receive almost no attention from scientific programmers.

As a result, solutions that appear natural to software engineers often don't work and might actually be counterproductive when applied in the

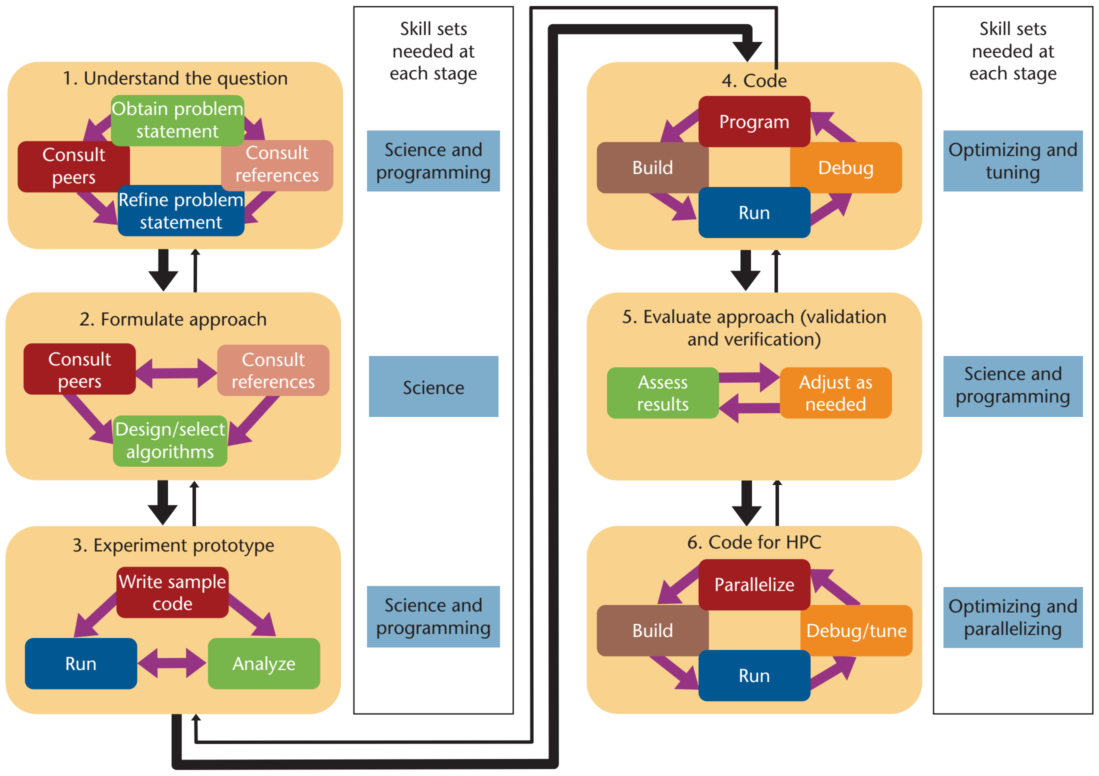

Figure 1. Programming workflow. As case studies of successful and failed projects show, realistic models must focus as much on the required skills as the time spent.

scientific programming context. This has resulted in antipathy between the two camps and fostered the view that software engineering has little to offer the scientific programming community.

In fact, software engineers don't understand the scientists' problems, most of which relate to topics that are barely taught in computer science curricula (such as floating-point arithmetic[10]). Even worse, scientific computing's distinct needs create critical conflicts among requirements—such as performance, hardware utilization, and portability—that are seldom experienced elsewhere.

Where Are the Productivity Bottlenecks?

To identify how and where current practices limit code development, we constructed an empirical workflow model of scientific programming activities. Among our data sources were extended observation of scientific programmers at work. For this, we drew on telemetric instrumentation of development tools, time-stamped journals written by the programmers, code inspections, and interviews.

Early analyses revealed that looking for bottlenecks in the traditional "edit-compile-run" loop offered little useful insight. For example, a coding period might be part of the mainstream code development, but it also might be an experiment—that is, downloading, modifying, and running a piece of code just to learn something. Alternately, time away from the keyboard (of which there's a great deal) might include a phone call for debugging help or additional insight into a numerical computation method.

As Figure 1 shows, our analysis also suggests that a realistic model must focus on the required skills as much as the time spent. This is a recurring theme in case studies of both successful and failed projects[9] and the key barrier to new programmers' effectiveness.[11]

Data from all of our studies revealed four general areas of effort.

- *Developing correct scientific programs.* This activity is characterized by both exploring an area of science and searching for relevant and tractable numerical methods. The outcome is a serial code that addresses the target problem and is scientifically sound.
- *Serial optimization and tuning.* This includes refining a serial code to ensure correctness and desired accuracy levels and improving serial performance (a critical predecessor to parallelization). This effort draws heavily on mathematical and scientific insight, such as knowing where to cut corners without decreasing accuracy.

- *Code parallelization and optimization.* This activity entails modifying code for parallel execution and requires additional optimization and tuning. Decomposing a numerical code for efficient distributed memory execution requires parallel programming expertise as well as mathematical and scientific insight. This phase can occupy half of a development cycle.
- *Porting and modifying existing parallel code.* This activity migrates a code to a new system and must maintain correctness and as much resource utilization and performance as possible.

This workflow model revealed extraordinary challenges in two general categories:

- *Manual programming.* This activity involves a tremendous amount of effort—programming and otherwise—that isn't intrinsic to the scientific problem domain, but rather emerges from the chosen technologies.
- *Expertise.* Most tasks demand multiple skill sets, including the domain science, numerical methods and programming, and parallelization.

These areas represent key bottlenecks in existing scientific programming processes that can't be removed without making fundamental changes to the way scientific codes have traditionally been developed.

What Is the "Expertise Gap"?

We revisited suggestions that an "expertise gap" lies at the heart of the crisis,[11] including one agency's assessment that its main problem was that it couldn't find enough people who could do the work. Drawing on our own case studies and five others gathered in collaboration with the research community,[9] we identified some common patterns.

In a representative case, one team member was recruited for specialized scientific knowledge, but had little programming expertise. A year spent collaborating with a scientific programming expert produced a code that was judged a failure; the outcome was blamed on insufficient communication between the two. Another project stalled at a later phase when no parallelization expert was available—a situation resolved only when a manager with this experience stepped in, leaving the management role (running interference with the customer, negotiating resources) to be backfilled.

Across all case studies, four distinct skill sets were seen as essential: domain science, scientific programming, scaling, and management. The skills are only useful, however, when they synchronize through communication and collaboration.

This is the expertise gap: it's difficult to find individuals or a set of individuals with the required mix of expertise in a specific application domain and experience on a specific hardware platform. This problem only worsens as machines become more complex. Further, expertise is developed through what amounts to an apprenticeship program that takes years: most professionals we interviewed saw their careers as based in science and engineering, with software playing the role of a necessary craft to be learned on the job or in graduate school.

Clearly, the expertise gap is a serious obstacle to productivity and limits the scientific programming community's ability to "scale out" to more application domains.[5]

Where Are the Tools?

"Better tools" should be an effective response to the expertise gap, making individual experts more productive. Unfortunately, the tools situation in the scientific programming community offers scant reason for optimism. Indeed, as we discovered, the unavailability of crucial tools (the "means of production" in economic terms) is often held responsible for a productivity decline.[6]

Pervasive complaints about tools included that they were hard to learn, don't scale, differ across platforms, are slow to appear on new platforms, and are poorly supported and too expensive. Several factors contribute to this state.

- The general computing community's high-productivity tools—such as the current generation of integrated development environments—are built around assumptions and practices that don't apply to scientific programming.
- Scientific code becomes tightly bound to the specialized tools used to build it, including compilers, libraries, and performance analyzers. Lifetime maintenance depends on tools that might not be available on every platform over the decades.
- Purchasing decisions and budgeting models are short-term and hardware-focused. Vendors have little incentive to supply quality software (operating systems, development tools, job management); this is particularly true in relation to long-term portability.
- The field is highly specialized and so are the tools. Because the market is small, prices are high, which discourages underfunded research institutions from buying the best tools.

- Tool investment by those who fund scientific missions is insufficient.[12] The small companies who market tools (usually drawing on technologies developed in research organizations) often fail.
- When smaller companies are sold, and especially when they're acquired by system vendors, successful cross-platform tools sometimes disappear from the market. One project we interviewed lost a year to recoding when a high-quality compiler disappeared in this fashion.

The result? Scientific programming projects view tools as a risk, not as a lever of productivity. Support models for this vital infrastructure are simply not aligned with its strategic value. To cope, scientific programming shops divert project resources into developing open source tools—not as an avenue to innovation as they might wish, but as a risk-mitigation strategy to ensure long-term access to the tools they already use.

Code Correctness and Productivity

While code correctness isn't traditionally considered a productivity issue, our multidisciplinary analysis suggests that it should be a major concern. We define increased (real) productivity here as increased production of useful scientific results for the resources expended. It follows trivially that producing more but scientifically incorrect results doesn't increase productivity and might in fact signify a decrease.

In conjunction with our analysis of workflow and the expertise gap, this led us to ask the question: *What levels of effort and expertise are required to assure the scientific correctness of codes?* As with other skills, we found that the effort involved, the length of time, and the expertise required result in a distinct productivity bottleneck. This challenge, simply put, is *trust* in computational outcomes' validity. If the results of scientific calculations can't be fully trusted, their value to the scientific community clearly diminishes. Indeed, this is now viewed as "the most serious limiting factor for computational science."[7]

When asked about strategies for building confidence in their codes, the scientific programmers we studied answered strictly in terms of the science. Typical responses? That the output "looked right" to the domain scientist, and confidence in the code grew over time. One scientist lamented the abandonment of an earlier code because it took them five years to start feeling "comfortable" with its replacement. A software engineer would seriously question a validation and verification strategy that relies on software output inspection and takes years to carry out.

Scientists are trained to manage threats to validity in experimental design but not in their codes. Software engineers have over the years developed many tools, technologies, and practices that contribute to increased software quality and reliability, but these appear largely unknown to scientific programmers.

Toward Solutions

Software engineering has much to offer toward solving this crisis, but only if the two sides can establish communication across a chasm of differing values and constraints. This will demand flexibility on both sides.

Software Engineering's Contribution

The software engineering community has developed a wide range of processes, methods, and techniques that help address productivity across the complete software life cycle. Some of these approaches could help resolve the current productivity bottlenecks, but only with effective collaboration and training. Others are conceptually relevant, but haven't been developed within scientific computing's assumptions and constraints. Such technologies often deliberately abstract away control over parameters that scientific programmers consider critical.

The upshot is that software engineers must return to their basic strategies and reapply them to develop and adapt to scientific computing's demands. In the near term, scientific programmers must be given tools to observe and control the performance-related parameters that matter to them. Successful longer-term advances will make those parameters less important. In every instance, software engineers must be prepared to make the case for these technologies using scientific programming's frame of reference, not just that of software engineering.

Software engineering also offers strategies for managing the expertise gap, though specific technologies must be reengineered to address scientific computing's needs and constraints.

Overall, we identified three basic software engineering strategies that are successful in other domains and could help ameliorate existing productivity bottlenecks by guiding the development of better scientific computing programming environments (languages, tools, instrumentation, and so on).

The first strategy is *automation*. The level-of-effort bottleneck can be improved only by asking the machine to perform more of the repetitive work. Parallelization is at the top of this list for scientific computing, followed by data layout and latency management, among others.

The second strategy is *abstraction*. The expertise gap arises from the inherent, irreducible difficulty in developing and maintaining expertise in three distinct disciplines over time. To address the bottleneck this entails, we must make it easier for scientists to write correct, efficient programs without also having to become experts in parallel programming and the idiosyncrasies of particular hardware platforms (such as complex memory models and high hardware failure rates). We must dedicate substantial automation toward providing scientifically relevant computational abstractions. Important higher-level abstractions will allow scientific programmers to express desired computations in ways that reflect the science and mathematics of the problem domain rather than the computing system.

The final strategy is *measurement*. Many new technologies will be needed here; to succeed, we must observe traditional parameters critical to scientific programming (such as processor utilization), along with feedback and continuous improvement in software development practices.

Scientific Programming's Role

Members of the scientific programming community have developed extraordinary mastery of numerical algorithms, optimization techniques, and problem decomposition, but they fail to appreciate how their goals have outgrown their software development practices. Addressing this requires two steps: investment and modernization.

The scientific programming community's insufficient *investment* in software infrastructure is ultimately responsible for many of the problems we observed.[12] Those who fund missions and lead the community must broaden their view of software development. This means optimizing across the entire development cycle—if not multiple cycles—rather than optimizing locally on next-run performance. It also means funding software infrastructure development, independent of specific vendors, which will add stability and create opportunity for the needed productivity growth.

Scientific programming must also *modernize*. Antipathy toward computer scientists must be abandoned and a fair hearing given to the case for improved practices. When the available technologies don't fit, scientific programmers must enlist software engineers to adapt them.

Promising Experiments

We ran a set of experiments that concretized the kind of productivity improvement that software engineering techniques can offer in the scientific computing realm. A professional scientific programmer rewrote several well-known HPC codes. The simplest were kernels from the NAS Parallel Benchmark suite (www.nas.nasa.gov/Resources/Software/npb.html), while the largest were the 14,000-line ASCI computational fluid dynamics code, Simplified Piecewise Parabolic Method (www.lcse.umn.edu/research/sppm/README.html), and the 7,000-line Gyrokinetic Toroidal Code code (http://gk.ps.uci.edu/GTC) for studying plasma microturbulence in fusion devices.

The objective of rewriting the code was to demonstrate improved programmability (that is, improved readability, maintainability, and verifiability). Specific changes included

- stripping out explicit MPI data distribution;
- removing manual optimizations;
- exercising modern languages features, such as Fortran 90 and array syntax;
- refactoring source code to increase the abstraction level;
- recasting source code to represent more clearly the underlying specification or mathematical algorithms;
- increasing code reuse, whether of user-written routines or available library functions ("copy and paste" lets you quickly write new source code, but it has a terrible impact on source bloat and maintainability);
- removing platform-specific source code;
- eliminating performance instrumentation, as its purpose is better served by performance analysis tools; and
- removing bug workarounds and other historical relics.

We achieved our experiments' objectives and obtained encouraging results in four areas: readability, compactness, expressivity of existing languages, and performance.

Readability. The rewritten code became considerably more readable and thus verifiable against algorithmic specifications. Figure 2 shows this at a small scale in the benchmark excerpt. We consulted with Stephane Ethier, who wrote the much larger GTC code, and he said he was impressed with the rewritten code's compactness and its expression of the underlying plasma physics.

Compactness. As Figure 3 shows, we reduced source code from three to 11 times its original volume across a range of cases. Approximately two times the size reduction was due to removal of explicit data decomposition and distribution; this result is consistent with other studies of MPI's contribution to code size. This simple reduction in code volume alone is significant, as the lifetime costs of software correlate strongly with code size.[13]

Expressiveness. Another striking result was that an existing programming language—in this case, Fortran 90 and its array syntax—proved to be quite expressive. Our first rewriting experiments weren't constrained to constructs or languages that could be compiled, or even that existed at all. That is, they were invitations to "program outside the box." Despite that freedom, the experiments tended to produce programs using existing language features—admittedly, these were sometimes "modern" features ahead of widespread adoption or mature compiler support.

We also recognized that high-level programming languages aren't always required. Sometimes, scientists want to "program in mathematics" for quick prototyping. For large-scale production programs, however, they want to program particular algorithms, incorporating the special insights that would otherwise leave many orders of performance magnitude behind.

Performance. The scientific programming community's first concern is cost in performance, including parallel speedup. In our experiments, the programmer found the performance shortfall to be rather tolerable—around two times for the bulk of considered cases. This held in cases up to even 100 threads, with support from commercial, automatically parallelizing compilers and large-scale shared memory hardware.[14] Many scientific programmers would see such a performance loss as quite serious. However, our studies convinced us that the performance loss is recoverable through strategic manual optimizations—or ideally from community investment in the supporting technologies. Software examples include

- interprocedural analysis, including for extracting concurrency;
- low-overhead work-sharing and stealing, for efficient concurrency;

```
call resid(u,v,r,n1,n2,n3,a,k)
callnorm2u3(r,n1,n2,n3,rnm2,rnmu,nx(lt),ny(lt),nz(lt))
old2 = rnm2
oldu = rnmu
do  it = 1,nit
    call mg3P(u,v,r,a,c,n1,n2,n3,k)
    call resid(u,v,r,n1,n2,n3,a,k)
enddo
call norm2u3(r,n1,n2,n3,rnm2,rnmu,nx(lt),ny(lt),nz(lt))
```
(a)

Each of the four iterations consists of the following two steps,
r = v - Au (evaluate residual)
u = u + Mkr (apply correction)
...
Start the clock before evaluating the residual for the first time, ...
Stop the clock after evaluating the norm of the final residual.

(b)

```
do iter = 1, niter
    r = v - A(u) ! evaluate residual
    u = u + M(r) ! apply correction
enddo
r = v - A(u)    ! evaluate residual
L2norm = sqrt(sum(r*r)/size(r))
```
(c)

Figure 2. Comparing the code. (a) The original Fortran 77 code and (b) the specification. (c) The rewritten Fortran 90 code is both compact and expressive of the underlying plasma physics.

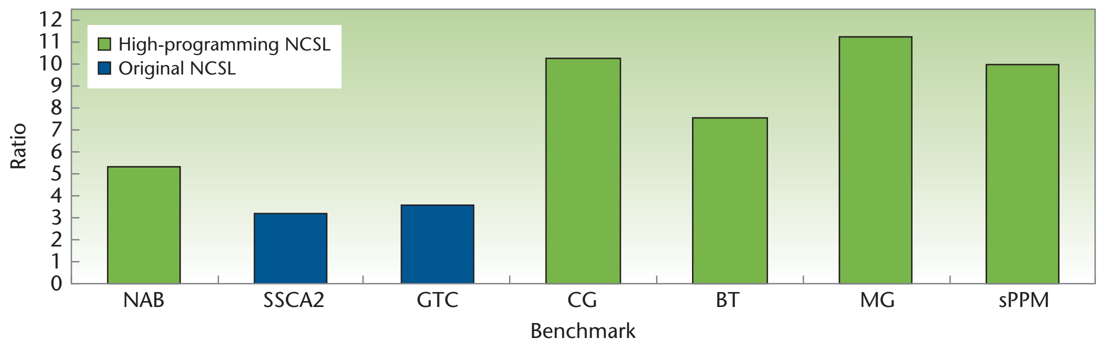

Figure 3. Reduction ratio of the original benchmark code size in noncomment source lines divided by rewritten benchmark code size in NCSL. Standards compared are Nucleic Acid Builder (NAB); the Scalable Synthetic Compact Application #2 (SSCA2) graph analysis benchmark; Gyrokinetic Toroidal Code (GTC); Conjugate Gradient (CG); Block Tridiagonal (BT); Multigrid (MG); and Simplified Piecewise Parabolic Method (sPPM).

- mature compiler support for modern language features, rather than just legacy or conservative programming practices; and
- runtime parallelization or concurrency checks.

Hardware examples include

- support for globally addressable memory;
- low-latency and high-throughput interconnects;
- latency-hiding techniques, such as scout threads and prefetch; and
- concurrency-supporting techniques such as transactional memory and active messages.

Application developers. Finally, support for higher programmability in scientific software must also come from the application developers themselves. Too often, software is developed expediently, at the expense of verifiability or long-term maintenance. Expressive, compact code requires programming for expressivity from the start, followed by ongoing perfective maintenance (rewriting the code to achieve a nonfunctional goal) over a program's lifetime.

Although we need more studies to quantify these results, we see an opportunity for concrete return-on-investment analysis that can make the case for dramatic change in scientific programming practices.

Scientific computing's productivity gridlock can be overcome, but only if programming practices change significantly.[2] Doing more of the same is a recipe for certain failure.

Success will require revisiting many common assumptions in software engineering and then re-engineering those solutions accordingly. It will also require far greater communication and collaboration between the software engineering and scientific computing communities. Our experience working in the latter suggests that this collaboration will be both fruitful and rewarding.

Acknowledgments

We're grateful to members of Sun's Extended Productivity Team and our HPCS program colleagues. We particularly thank our case study collaborators, including Douglass Post, Richard Kendall, and Walter Tichy, as well as Tom Nash and Philip Johnson, whose work deepened our understanding of the productivity problem. Thanks also to our reviewers whose thoughtful comments have significantly improved this paper. DARPA supported this work under contract number NBCH3039002.

References

1. C. Holland, *DoD Research and Development Agenda for High Productivity Computing Systems*, Pentagon white paper, US Defense Dept., 2001.
2. M. Van De Vanter et al., *Productive Petascale Computing: Requirements, Hardware, and Software*, tech. report TR-2009-183, Sun Microsystems, 2009.
3. D.E. Perry, N.A. Staudenmayer, and L.G. Votta, "Understanding and Improving Time Usage in Software Development," A. Wolf and A. Fuggetta, eds., *Software Process*, vol. 5, John Wiley & Sons, 1995, pp. 111–135.
4. S. Squires, M. Van de Vanter, and L. Votta, "Software Productivity Research in High Performance Computing," *CTWatch Quarterly*, vol. 2, no. 4A, 2006; www.ctwatch.org/quarterly/articles/2006/11/software-productivity-research-in-high-performance-computing.
5. S. Squires, M. Van De Vanter, and L. Votta, "Yes, There Is an 'Expertise Gap' in HPC Applications Development," *Proc. 3rd Int'l Workshop on Productivity and Performance in High-End Computing* (PPHEC'06), IEEE CS Press, 2006, pp. 5–10.
6. M. Van De Vanter, D. Post, and M. Zosel, "HPC Needs a Tool Strategy," *Proc. 2nd Int'l Workshop Software Eng. High-Performance Computing Systems Applications*, ACM Press, 2005, pp. 55–59.
7. D. Post and L. Votta, "Computational Science Demands a New Paradigm," *Physics Today*, vol. 58, no. 1, 2005, pp. 35–41.
8. D. Kelly, "A Software Chasm: Software Engineering and Scientific Computing," *IEEE Software*, vol. 24, no. 6, 2007, pp. 120–119.
9. 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.
10. D. Goldberg, "What Every Computer Scientist Should Know about Floating-Point Arithmetic," *ACM Computing Surveys*, vol. 23, no. 1, 1991, pp. 5–48.
11. V. Sarkar, C. Williams, and K. Ebcioglu, "Application Development Productivity Challenges for High-End Computing," *Proc. 1st Workshop Productivity and Performance High-End Computing*, ACM Press, 2004, pp. 14–18.
12. S. Graham, M. Snir, and C. Patterson, eds., *Getting Up To Speed: The Future of Supercomputing,* Nat'l Academies Press, 2005.
13. B.W. Boehm, *Software Engineering Economics,* Prentice Hall, 1981.
14. E. Loh, M. Van De Vanter, and L. Votta, "Can Software Engineering Solve the HPCS Problem?" *Proc. 2nd Int'l Workshop Software Eng. High-Performance Computing Systems Applications,* ACM Press, 2005, pp. 27–31.

Stuart Faulk *is an associate research professor in the Computer Science Department at the University of Oregon. His research interests include software productivity, software processes, requirements engineering, and software product lines. Faulk has a PhD in computer science from the University of North Carolina at Chapel Hill. Contact him at faulk@cs.uoregon.edu.*

Eugene Loh *is a senior staff engineer at Sun Microsystems, where he focuses on performance analysis within high-performance computing and has studied programming and performance issues as part of Sun's HPCS Phase II work. His research interests include performance analysis of HPC applications. Loh has a PhD in physics from the University of California, Santa Barbara. Contact him at eugene.loh@sun.com.*

Susan Squires *is executive director of customer insight research at Tactics, LLC, and a practicing anthropologist with wide experience in customer research, strategic planning, and program management. Her research interests include technology usability, scientific workflow modeling, and the intersection of technology use and workgroup organization. Squires has a PhD in anthropology from Boston University and is a fellow of the Society for Applied Anthropology. Contact her at susan.squires@acelere.net.*

Michael L. Van De Vanter *is a senior staff engineer at Sun Microsystems, where he was a member of Sun's HPCS Core Productivity team and principal investigator of the Jackpot Project. His research interests include software development technologies, practices, and tools. Van De Vanter has a PhD in computer science from the University of California Berkeley, and is a member of the IEEE Computer Society and the ACM. Contact him at mlvdv@ieee.org.*

Lawrence G. Votta *is a consultant for software fault tolerance and productivity at Brincos. He was a Sun Microsystems Distinguished Engineer, working to improve software and system reliability and serving as a principle investigator for the DARPA High Productivity Computing System (HPCS) initiative and leading the productivity analysis team. His research interests include high-availability computing and empirical software engineering. Votta has a PhD in physics from the Massachusetts Institute of Technology. Contact him at votta@alum.mit.edu.*