Where's the Real Bottleneck in Scientific Computing?

Gregory V. Wilson

Scientists would do well to pick up some tools widely used in the software industry

When I first started doing computational science in 1986, a new generation of fast, cheap chips had just ushered in the current era of low-cost supercomputers, in which multiple processors work in parallel on a single problem. Suddenly, it seemed as though everyone who took number crunching seriously was rewriting his or her software to take advantage of these new machines. Sure, it hurt—the compilers that translated programs to run on parallel computers were flaky, debugging tools were nonexistent, and thinking about how to solve problems in parallel was often like trying to solve a thousand crossword puzzles at once—but the potential payoff seemed enormous. Many investigators were positive that within a few years, computer modeling would let scientists investigate a whole range of phenomena that were too big, too small, too fast, too slow, too dangerous or too complicated to examine in the lab or to analyze with pencil and paper.

But by the mid-1990s, I had a nagging feeling that something was wrong. For every successful simulation of global climate, there were a dozen or more groups struggling just to get their program to run. Their work was never quite ready to showcase at conferences or on the cover of their local supercomputing center's newsletter. Many struggled on for months or years, tweaking and tinkering until their code did something more interesting than grinding to a halt or dividing by zero. For some reason, getting to computational heaven was taking a lot longer than expected.

I therefore started asking scientists how they wrote their programs. The answers were sobering. Whereas a few knew more than most of the commercial software developers I'd worked with, the overwhelming majority were still using ancient text editors like Vi and Notepad, sharing files with colleagues by emailing them around and testing by, well, actually, not testing their programs systematically at all.

I finally asked a friend who was pursuing a doctorate in particle physics why he insisted on doing everything the hard way. Why not use an integrated development environment with a symbolic debugger? Why not write unit tests? Why not use a version-control system? His answer was, "What's a version-control system?"

A version-control system, I explained, is a piece of software that monitors changes to files—programs, Web pages, grant proposals and pretty much anything else. It works like the "undo" button on your favorite editor: At any

Greg Wilson is an adjunct professor of computer science at the University of Toronto. His course material is available at http://www.third-bit.com/swc. On February 17, as part of the 2006 Annual Meeting of the American Association for the Advancement of Science in St. Louis, Wilson will be running a workshop on scientific programming skills, how investigators should acquire them, and the changes needed in publication and tenure procedures to ensure quality in computational work. Address: Room 3230, Bahen Centre for Information Technology, University of Toronto, Toronto, Ontario, M5S 2E4. Internet: gvwilson@cs.utoronto.ca

point, you can go back to an older version of the file or see the differences between the way the file was then and the way it is now. You can also determine who else has edited the file or find conflicts between their changes and the ones you've just made. Version control is as fundamental to programming as accurate notes about lab procedures are to experimental science. It's what lets you say, "This is how I produced these results," rather than, "Um, I think we were using the new algorithm for that graph—I mean, the old new algorithm, not the new new algorithm."

My friend was intelligent and intimately familiar with the problems of writing large programs—he had inherited more than 100,000 lines of computer code and had already added 20,000 more. Discovering that he didn't even know what version control meant was like finding a chemist who didn't realize she needed to clean her test tubes between experiments. It wasn't a happy conversation for him either. Halfway through my explanation, he sighed and said, "Couldn't you have told me this three years ago?"

As the Twig Is Bent…

Once I knew to look, I saw this "computational illiteracy" everywhere. Most scientists had simply never been shown how to program efficiently. After a generic freshman programming course in C or Java, and possibly a course on statistics or numerical methods in their junior or senior year, they were expected to discover or reinvent everything else themselves, which is about as reasonable as showing someone how to differentiate polynomials and then telling them to go and do some tensor calculus.

Yes, the relevant information was all on the Web, but it was, and is, scattered across hundreds of different sites. More important, people would have to invest months or years acquiring background knowledge before they could make sense of it all. As another physicist (somewhat older and more cynical than my friend) said to me when I suggested that he take a couple of weeks and learn some Perl, "Sure, just as soon as you take a couple of weeks and learn some quantum chromodynamics so that you can do *my* job."

His comment points at another reason why many scientists haven't adopted better working practices. After being run over by one bandwagon after another, these investigators are justifiably skeptical when someone says, "I'm from computer science, and I'm here to help you." From object-oriented languages to today's craze for "agile" programming, scientists have suffered through one fad after another without their lives becoming noticeably better.

Scientists are also often frustrated by the "accidental complexity" of what computer science has to offer. For example, every modern programming language provides a library for regular expressions, which are patterns used to find data in text files. However, each language's rules for how those patterns actually work are slightly different. When something as fundamental as the Unix operating system itself has three or four slightly different notations for the same concept, it's no wonder that so many scientists throw up their hands in despair and stick to lowest common denominators.

Just how big an impact is the lack of programming savvy among scientists having? To get a handle on the answer, consider a variation on one of the fundamental rules of computer architecture, known as Amdahl's Law. Suppose that it takes six months to write and debug a program that then has to run for another six months on today's hardware to generate publishable results. Even an infinitely fast computer (perhaps one thrown backward in time by some future physics experiment gone wrong) would only cut the mean time between publications in half, because it would only eliminate one restriction in the pipeline. Increasingly, the real limit on what computational scientists can accomplish is how quickly and reliably they can translate their ideas into working code.

A Little Knowledge

In 1998, Brent Gorda (now at Lawrence Livermore National Laboratory) and I started trying to address this issue by teaching a short course on software-development skills to scientists at Los Alamos National Laboratory. Our aim wasn't to turn LANL's physicists and metallurgists into computer scientists. Instead, we wanted to show them the 10 percent of modern software engineering that would handle 90 percent of their needs.

The first few rounds had their ups and downs, but from what participants said, and from what they did after the course was over, it was clear that we were on the right track. A few techniques, and an introduction to the tools that supported them, could save scientists immense frustration. What's more, we found that most scientists were very open to these ideas, which probably shouldn't have surprised us as much as it did. After all, the importance of being methodical had been drilled into them from their first undergraduate lab.

Six years and one dot-com boom later, I received funding from the Python Software Foundation to bring that course up to date and to make it available on the Web under an open license so that anyone who wants to use it is free to do so. It covers tools and working practices that can improve both the quality of what scientific programmers produce, and the speed with which they produce it, so that they can spend less time wrestling with their programs and more doing their research. Topics include version control, automating repetitive tasks, systematic testing, coding style and reading code, some basic data crunching and Web programming, and a quick survey of how to manage development in a small, geographically distributed team. None of this is rocket science—it's just the programming equivalent of knowing how to titrate a solution or calibrate an oscilloscope.

On the Hard Drives of Giants

Science is much more than just a body of knowledge. It's a way of doing things that lets people separated by oceans, decades, languages and ideologies build on one another's discoveries. Computers are playing an ever-larger role in research with every passing year, but few scientific programs meet the methodological standards that pioneers like Lavoisier and Faraday set for experimental science more than 200 years ago.

Better education is obviously key to closing this gap, but it won't be enough on its own. Journals need to start insisting that scientists' computational work meet the same quality and reproducibility standards as their laboratory work. At the same time, we urgently need more journals willing to publish descriptions of how scientists develop software, and how that software functions. Faster chips and more sophisticated algorithms aren't enough—if we really want computational science to come into its own, we have to tackle the bottleneck between our ears.