The ASC-Alliance Projects: A Case Study of Large-Scale Parallel Scientific Code Development

Lorin Hochstein, University of Nebraska-Lincoln
Victor R. Basili, University of Maryland and Fraunhofer Center for Experimental Software Engineering

Computational scientists face many challenges when developing software that runs on large-scale parallel machines. However, software-engineering researchers haven't studied their software development processes in much detail. To better understand the nature of software development in this context, the authors examined five large-scale computational science software projects operated at the five ASC-Alliance centers.

Computational scientists use computers to simulate physical phenomena in situations where experimentation would be prohibitively expensive or impossible. Advancing scientific research depends on these scientists' developing software productively. However, the software development process in this domain differs from other domains. For instance, scientific software can be computationally demanding and require the most powerful machines. These machines, referred to as supercomputers or high-end computing systems, present unique challenges to software development.

To learn more about developing software to run on HEC systems, we studied five large software projects that develop such computer programs—referred to as *codes* in the HEC community. These projects are carried out at the five Advanced Simulation and Computing-Alliance research centers, with each project addressing a different computational science problem and each center having access to large-scale HEC systems at various supercomputing centers.

We interviewed high-level ASC-Alliance project participants involved in project management, software architecture, or software integration. Our goals were to identify challenges the scientists face and to characterize product, project organization, and process in terms of using and developing software.

ASC-ALLIANCE CENTERS

The US Department of Energy's National Nuclear Security Administration formed the five ASC-Alliance centers around 1997 to develop computational simulation as a credible scientific-research method. The five centers are

- the University of Utah's Center for Simulation of Accidental Fires and Explosions, which simulates large fires and embedded explosives, as Figure 1 shows;
- the University of Illinois at Urbana-Champaign's Center for the Simulation of Advanced Rockets, which simulates solid-propellant rockets, as Figure 2 shows;
- the University of Chicago's Center for Astrophysical Thermonuclear Flashes, which simulates stars' thermonuclear burn, as Figure 3 shows;
- Caltech's Center for Simulating the Dynamic Response of Materials, which simulates materials' response to strong shocks, as Figure 4 shows; and
- Stanford University's Center for Integrated Turbulence Simulations, which simulates full-scale jet engines, as Figure 5 shows.

Published by the IEEE Computer Society

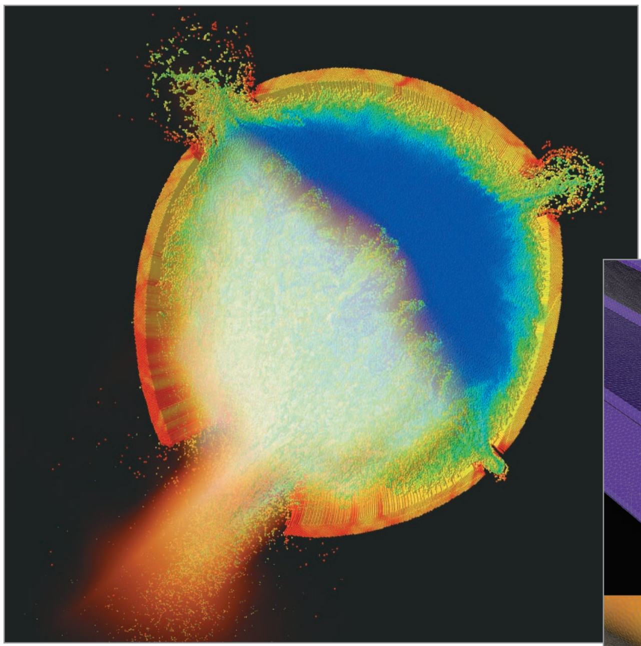

Figure 1. Image from the Center for Simulation of Accidental Fires and Explosions, based at the University of Utah, shows a simulation of a metal container rupturing after the contained explosive ignited and pressurized the container when it transitioned from a solid to gaseous state.

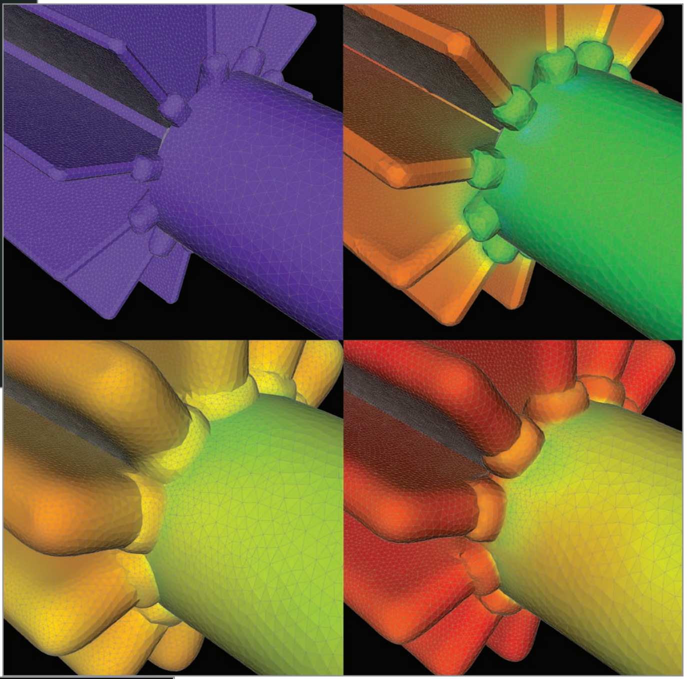

Figure 2. Images from the Center for the Simulation of Advanced Rockets, based at the University of Illinois at Urbana-Champaign. These images show a simulation of the space shuttle's reusable solid rocket motor as the propellant burns back. The gas pressure (indicated by color) in the fluid domain near the motor's head end is shown at four different times.

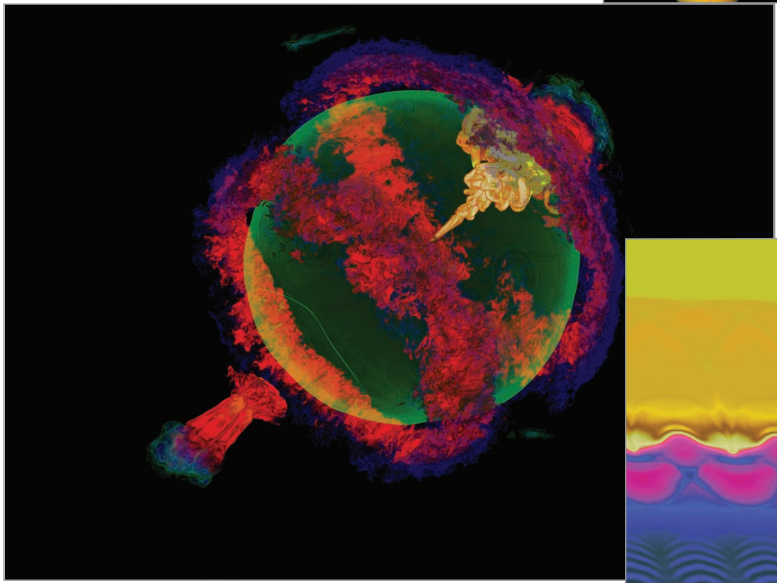

Figure 3. Image from the Center for Astrophysical Thermonuclear Flashes, based at the University of Chicago, shows a simulation of a Type Ia supernova two seconds after the ignition of a nuclear flame near the center of a white dwarf.

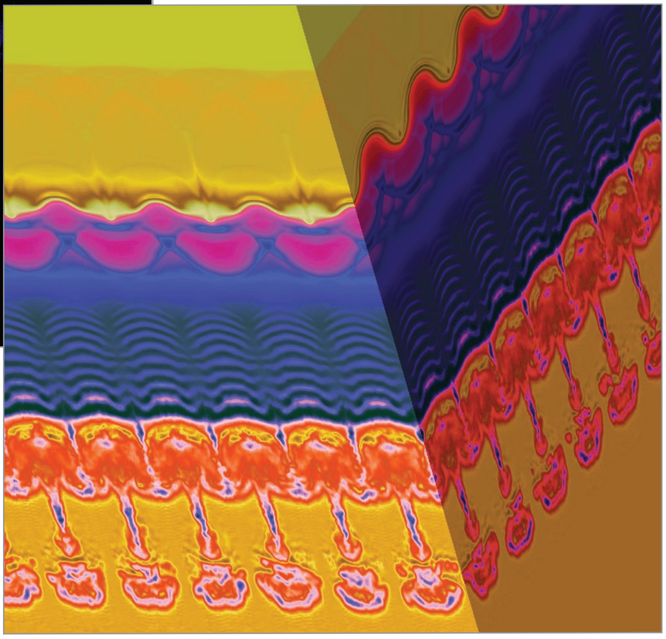

Figure 4. Image from the Center for Simulating the Dynamic Response of Materials, based at Caltech, shows a simulation of the response of materials to strong shocks. The figure shows fluid density during turbulent mixing in a shock tube experiment as high-speed waves, shocks, and rarefactions travel between the walls of the tube.

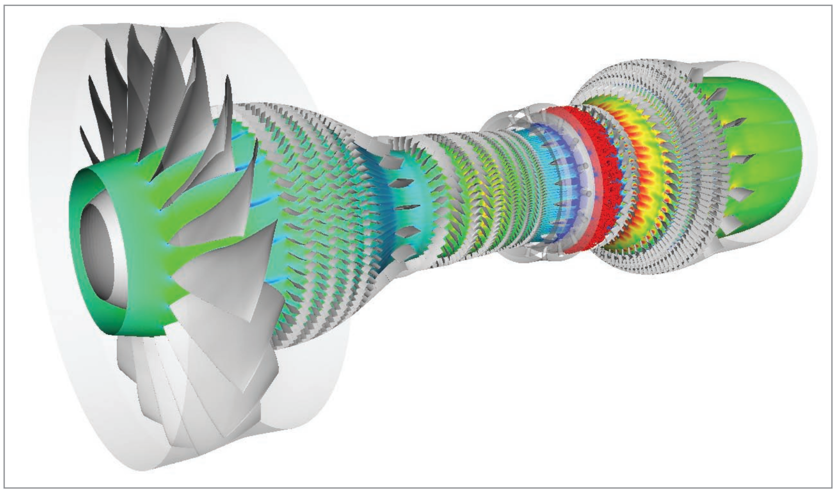

Figure 5. Image from Stanford University's Center for Integrated Turbulence Simulations depicts a simulation of a Pratt & Whitney jet engine showing instantaneous axial velocity at midspan.

The major project at each center focuses on solving one particular scientific problem by developing multiphysics, coupled applications. This refers to the simulation of different aspects of physical phenomena—for example, solid mechanics, fluid mechanics, or combustion— that are "coupled" to form a single simulation.

GOALS AND METHODOLOGY

Our goals were to characterize which scientific programming activities are time-consuming and problematic, identify common problems scientific programmers face, and assess the impact of software technologies on scientists' development time. We conducted this study within the context of DARPA's High Productivity Computing Systems (HPCS) project (www.highproductivity.org), which is aimed at improving computational scientists' productivity by developing new machine architectures, new parallel-programming languages, and other technologies.

In our earlier work, we ran controlled experiments to evaluate the effect of parallel-programming language on programmer effort and program performance, using students from graduate-level parallel-computing courses.[1] However, without empirical data on how scientific codes are developed, we had no larger context for interpreting our results. In particular, we didn't know whether a new parallel-programming language would address the major problems developers faced, or whether they'd adopt a new language if given the opportunity.

To begin our most recent study, we sent a questionnaire requesting basic project information from each center. Next, we conducted telephone interviews with one or two technical leads on each project. From these interviews, we generated summary documents that we sent back to the technical leads for review and corrections.

SOFTWARE CHARACTERISTICS

While our main object of study was the scientists' software development process, we first wanted to characterize the product they were working on to provide context for their software environment. We asked about the software's attributes (code size, organizational structure, and degree of code reuse via libraries), and the intended machine target (what kinds of machines the codes are intended to run on).

Attributes

The codes range from 100,000 to 500,000 lines. Most are written as a mixture of C/C++ and Fortran, with one code a pure Fortran implementation. One code uses a Python scripting layer that provides an interface for running the application. With one exception, core elements of these projects evolved from preexisting codes. All codes use the message-passing interface (MPI) library to achieve parallelism. In addition, each code uses external libraries for features such as

- I/O (HDF, NetCDF, CGNS, or Panda);
- mesh operations, including adaptive mesh refinement (ParaMesh, Mesquite, Metis, MeshSim, or SAMRAI);
- computational geometry (CGAL);
- linear algebra (BLAS, LAPACK), and
- tools for solving sparse linear systems and systems modeled by partial differential equations (PETSc, Hypre, or Clawpack).

While these codes use parallel libraries that sit atop MPI, developers still had to write raw MPI code to achieve desired functionality. Therefore, they dealt with the additional complexities of writing message-passing applications. Some codes use a layered approach that hides the details of message-passing, so a programmer can add functionality without writing MPI code. However, programmers had to write these abstraction layers from scratch.

Each code is organized into independent subsystems that individuals or small groups maintain. All codes use a component-based architecture to minimize coupling between individual subsystems. In several cases, these independent subsystems are almost like separate projects. They can run as stand-alone applications and might incorporate new features independent of the larger, coupled application. Since most codes involve multiple programming languages, there are language interoperability issues. The one exception, a pure Fortran application, once used a Python framework to drive the application but it was abandoned because of the difficulty in porting a hybrid Python/Fortran application to multiple platforms.

One project's component framework was built around the Common Component Architecture, a community effort to simplify building such multilanguage, coupled codes. In that case, the chief software architect was an early adopter of this technology and is actively involved with the larger CCA effort. The other projects developed their own communication frameworks.

Machine target

The codes are designed to run on "flat" MPI-based machines, where communication takes place through message-passing, even if some processes share physical memory. While all the codes currently run on clusters of symmetric multiprocessors, none has been explicitly optimized to take advantage of the SMP nodes. The developers assume that the vendor MPI implementations are efficient enough that optimizing for SMP nodes won't yield large performance improvements. Tuning for a specific architecture is considered a poor use of resources. The investment required to gain expertise in a particular architecture is too great given that new architectures appear every six months.

Two projects groups experimented in the past with improving performance on clusters of SMPs by using OpenMP to leverage parallelism within nodes and MPI to leverage parallelism across nodes. Results were mixed. One group found that a pure MPI implementation was competitive with a hybrid MPI-OpenMP approach, and the other observed increased performance when incorporating OpenMP but hasn't followed up on this work due to other priorities.

PROJECT ORGANIZATION

The scientists must coordinate their efforts, since these projects involve more than one person. We wanted to understand the organizational structure, the staff, and their configuration-management process. We were looking for similarities and differences with software projects in other domains, and we wanted to determine whether the scientists encountered any domain-specific issues from a project-management point of view.

Organizational structure

Each project is divided into groups that focus on different aspects of the problem. This division is reflected in the code, where the software is partitioned into independent subsystems, and one group owns each subsystem. Each subsystem has one or two chief programmers who understand the subsystem in depth and are responsible for it. These chief programmers make the majority of the code changes. Each project also has either a chief software architect or a group responsible for the integration code.

Development is compartmentalized and the groups are relatively independent. Integrated code-development meetings are held once a week to let core developers discuss issues such as coordinating code changes that will affect more than one module.

The software is partitioned into independent subsystems, and one group owns each subsystem.

Staff

About 75 people are actively involved on a given project. Ten to 25 are core developers who routinely contribute code. The developers consist of professionals, professional staff members with MSs and PhDs, postdocs, and graduate students. Their backgrounds are in physics, chemistry, applied math, engineering (mechanical, civil, aerospace, or chemical), and computer science. The programmers have from 5 to 25 years of sequential programming experience, and 0 to 15 years of parallel-programming experience. Graduate students also work on the code as part of their research, though they aren't core developers.

Configuration management

The projects use version-control systems such as CVS and Subversion to coordinate changes to the code, and all have integrated version control into their development process. No projects have a formal process for approving code before it's checked into the repository. Instead, there's agreement that test cases should pass before commits are made to the repository. Developers are individually responsible for performing any necessary unit, standalone, and integration testing. On one project, developers are automatically notified by e-mail whenever code is checked in to the repository so that they're aware of recent modifications that might affect them.

Since all codes are actively used for scientific research and development, the projects must allow the developers to modify the code while ensuring that a stable version is always available. Therefore, all projects maintain both stable and development versions of the code.

Only one project has a formal bug-tracking system in active use. On the other projects, defect tracking is accomplished through wikis and informal communication among project members. Some projects have attempted to introduce defect-tracking systems, but developers didn't adopt them.

SOFTWARE USAGE

Our study focused mainly on software development. However, we also wanted to get a sense of how the software was used, and who was using it. Since requirements are a major issue in other domains of software engineering, and user needs drive requirements, we wanted to understand the user's role in this domain. In addition, we wanted to understand execution times. We didn't know how long these types of programs took to run, and we believed that large execution times were a major obstacle to programmer productivity. We wanted to understand the entire process of how the software was used, from setting up the input to examining the output.

The main users of the codes are research scientists who are the active developers. Some students also use the software for their own scientific research, and aren't active in the code development, but these efforts aren't the centers' primary concern. Some codes have external users who might modify the programs to suit their own needs.

Characterizing execution times is difficult because they vary enormously depending upon the size of the problem. Typical runs are on the order of 10 to 100 hours.

Most projects use configuration files for specifying program parameters, with two exceptions: One project uses an interactive Python-based scripting interface, and another provides a programmatic Fortran interface for specifying the simulation's initial conditions. Some projects have expressed interest in developing a graphical interface to simplify the task of setting up the input for a run.

For some projects, generating inputs is time-consuming. Some codes simulate systems with intricate geometries (for example, the space shuttle), which are modeled as unstructured meshes. Generating the mesh for an input can take an experienced user from half an hour to weeks or months. In one case, a user spent a year generating a mesh for input. Determining whether a given mesh is of sufficient quality is an active area of research.

Users apply visualization tools to examine the simulations' output. The projects use a mix of visualization tools developed in-house (Flashview, Rocketeer, and SCIRun) or by third parties (IDL, TecPlot, EnSight, ParaView, OpenDX, Matlab, Iris Explorer, and VisIt).

DEVELOPMENT ACTIVITIES

The developers engage in different activities during the course of development. We asked for details about adding new features and testing, tuning, debugging, and porting the code.

Adding new features

Each center plans to run a major set of simulations each year. These simulations drive an implementation plan that determines needed new features. Scientists can explore research avenues, but the implementation plan sets the overall direction. Demand from large outside user bases can also drive new features.

New features can be classified into two categories:

- those localized within an individual subsystem (low-level change), or
- those involving changes across subsystems (high-level change).

Low-level changes are administered solely by owners of the subsystem being modified and require no communication across groups. High-level changes require some degree of coordination.

Since the projects have been in operation for almost a decade, the code bases are all mature and researchers are applying them to do real science. While enhancements to existing subsystems continue, few new subsystems are planned. Most modules have satisfactory parallel performance, with the exception of very new modules and modules where efficient parallelization is still an open research problem (for example, adaptive mesh refinement).

All projects use a suite of regression tests to catch any errors programmers introduce while modifying the code.

In some projects, the developers don't have to write code explicitly in parallel but instead build atop a parallel infrastructure that abstracts away the parallelization details. Other projects require the developers to program directly to the MPI library.

Testing

All projects use a suite of regression tests to catch any errors programmers introduce while modifying the code. Some projects have an automated system for running regression tests, and others run the regression tests manually. One project requires new students to run the regression tests as part of their learning process.

Testing new algorithms. Testing a new algorithm is challenging in this environment. It's not sufficient to define simple test cases where modules are fed known inputs and checked against expected outputs. Rather, the researchers evaluate the algorithms in terms of stability, accuracy, speed, and linear scalability. A module is considered to be functioning correctly if, for the class of inputs, the quality of the module's output is sufficient to let it be coupled with other modules and produce coupled applications. Since the inputs of interest change as scientists try more complex simulations, an algorithm that is acceptable today might not be acceptable tomorrow.

Therefore, the testing process is different from other software domains because the focus is on identifying algorithmic defects (evaluating the algorithm's quality) rather than on coding defects (errors in implementing the algorithm in the source code). Finding and fixing algorithmic defects is much more challenging than finding and fixing coding defects.

Testing algorithm quality. Testing the quality of algorithms involves qualitative analysis to determine how the algorithm behaves. There are different strategies for testing an algorithm, depending on the nature of the problem—for example, checking if certain quantities are exactly or approximately conserved, if symmetry properties hold, and against known analytical solutions. Some projects work with numerical analysts who can provide mathematical guarantees about certain aspects of the code such as stability or that certain positive quantities such as energy can't diverge.

In general, the developers don't know whether an algorithm solves an equation correctly until certain requirements are passed. For example, a module might seem to be performing correctly in isolation, but when used in a coupled application, it might behave in unexpected ways.

This interactive testing process requires a substantial amount of effort and expertise. Since many of the developers are postdocs and graduate students without extensive experience, the testing process involves much guidance from senior people who understand the broader scope of the physics and software.

On at least one project, the codes are constantly in flux as new algorithms are continually evaluated, changing the core components of the code.

Tuning

Tuning activity occurs when the developers discover that the software is executing much slower than expected. The software might need tuning when it's being ported to a new platform, if major changes to the software architecture have caused performance penalties, or simply because changes made to a particular subsystem create a bottleneck. At least one project uses tuning specialists—developers skilled at identifying and fixing performance bottlenecks. One particular tuner comes from a local computer science group that develops performance-analysis tools.

Platform range. Since one of the project goals is to develop algorithms that will last across many machine lifetimes, it's not seen as productive to try to maximize the performance on any particular platform. Instead, code changes are made that will improve performance on a wide range of platforms. In addition, on at least one project, the codes are constantly in flux as new algorithms are continually evaluated, changing the core components of the code. If there were many machine-specific optimizations in the code, understanding the code would be much more difficult, which would increase maintenance effort.

For a given application, a considerable amount of tuning is needed to achieve reasonable performance on a new platform. This tuning process is mostly about determining data-set size, number of processors, and which processors should be assigned which tasks. While individual projects don't focus on maximizing performance on any one system, they occasionally can take advantage of a team of third-party experts who can achieve a large speedup on a particular system.

External tools. Developers do use externally developed profiling tools such as Jumpshot, SpeedShop, or Shark. However, on some of the codes, external profiling tools couldn't deal with an application written in multiple languages. In addition, some codes contain their own profiling routines. Some developers find these tools useful, but others say they're familiar enough with the code that these profiling tools don't reveal new information.

There are ongoing efforts to improve performance through development of new algorithms, such as new adaptive mesh refinement algorithms. However, the developers view these efforts as new functionality rather than tuning.

Debugging

All the projects use TotalView, a popular parallel debugger. Sequential debugging tools such as Purify and Ensure are also used, but they are useful only if the failure can be reproduced when running the program on a single processor. Trace statements for debugging are common, although they're difficult to interpret when the program is running on many processors. The developers also examine the simulation outputs with visualization tools to help identify defects.

Usage patterns. Developers described several usage patterns for applying the tools to localize defects. One common pattern is to use a debugger to produce a stack trace, which is then used to determine where to insert print statements into the code. Another common debugging pattern is to try to reproduce a defect when running the program on a smaller simulation, as tools such as TotalView and gdb can be used effectively on a moderate number of CPUs.

Existing tools don't provide any extra assistance for some types of defects. Typical debugging tools aren't appropriate for algorithmic debugging, which takes up most of the debugging time.

Batching. Large machines are generally batch-scheduled, which makes debugging more difficult. Since the debugging process typically involves frequent rerunning of the code, running under a batch queue can take a week, rather than hours, with a dedicated machine running jobs interactively. The national laboratories give ASC-Alliance centers partial or full interactive use of a particular machine for a few weekends each year. The 60 hours of total compute time lets the centers run very large jobs and do large-scale debugging. Debugging runs that involve smaller numbers of processors are done internally.

Porting

Porting commonly occurs when the Department of Energy buys a new machine that becomes available for project use. Porting might be necessary due to a software upgrade on a system in use. It might take from a day to several weeks to port to a new system, depending on the code and maturity of the new platform's development tools.

Porting requires a nontrivial amount of effort because when a new platform is released, the code can't simply be recompiled and run. Much of the porting effort is spent on a large number of small details. For example, developers need to modify the build scripts (makefiles) to accommodate differences in the new system's development tools. Immature compilers on new systems are another source of porting effort. For example, one project spent a year getting the code to run on one machine because of C++ issues. While workarounds solved some problems, others required waiting for the vendor to fix the compiler. One project abandoned Python to reduce porting effort.

Although porting is only a small part of the total effort, it can involve a great deal of work in a very focused time—for example, several people working for a month. Participants perceive porting as requiring an unwarranted amount of effort. Some projects have broken compilers and MPI codes on every platform. Two projects didn't port their code to ASCI Red because the Fortran compiler didn't properly support needed features, and it wasn't worth the effort to work around these problems to get the code to run.

Effort distribution and bottlenecks

Developers report spending 75-95 percent of their development time on new features and testing. Tuning and porting take 1-10 percent of the total development time. However, the distribution of effort varies depending on the project's development phase. Earlier in the projects, developers use a different effort distribution to design new data structures to handle a particular class of physics problems than they do later, when they're trying to modify data structures for new areas. The projects occasionally undergo large-scale rearchitecting to better incorporate new features, and this also changes the effort distribution.

Verification and validation is a common bottleneck, in particular, debugging parallel algorithms. Defects that are only manifest in more complex execution environments where observation is more difficult for the programmer make debugging a challenge. For example, a code might run perfectly on 32 processors but fail on 64 processors. Or a subsystem might work well when executing independently, but doesn't behave as expected when interacting with another subsystem.

Other bottlenecks include generating input (CAD modeling, mesh generation), expressing algorithms in parallel, performing production runs (obtaining time to run on large machines), and understanding old code (when rearchitecting).

GENERAL OBSERVATIONS

Based on our interviews, we made some general observations about software engineering in this domain. It surprised us how challenging it is to validate this type of software. We're also interested in the role MPI played, because of the HPCS project's goals to develop alternative parallel-programming languages. In particular, we're interested in understanding an alternative to MPI, either in the form of a library or framework that encapsulates MPI, or another parallel-programming language. Finally, we're interested in the scientists' opinions on productivity, since one of our ultimate goals is to improve programmer productivity.

Despite MPI's being the most appropriate technology for the job, there's widespread dissatisfaction with it.

Validation

Validating the codes is a formidable challenge. A validation study is typically a research project or thesis. A student will choose a problem with an interesting set of associated experimental data and then identify how to simulate it. Because of the effort involved in such studies, it's not feasible to validate every new algorithm with an experiment.

MPI

All projects make extensive use of MPI to achieve parallelism. MPI is a standard library that's efficiently implemented on all types of platforms: It's the only technology that can run 10,000 processor jobs and allow the program to be run on a workstation or world-class machine without tweaking the code in significant ways. In addition, the algorithms the projects employ lend themselves to domain decomposition.

Despite MPI's being the most appropriate technology for the job, there's widespread dissatisfaction with it. One project member referred to it as "nothing more than high-level assembly language," where the programmer is responsible for all memory management and process signaling. The additional work MPI requires was identified as a large barrier to productivity for a project starting from scratch. However, one project member mentioned that exposing these low-level details to the programmer is an advantage of MPI, because it gives the programmers more control over what happens in the code.

One advantage is that it's possible to write MPI code knowing very few functions, so it's easy to teach. While teaching people to use MPI isn't hard, teaching people to write MPI effectively is extremely difficult. This fact distinguishes first-year graduate students from developers who've been at a center for four or five years.

Libraries and frameworks

All the projects used libraries built atop MPI. However, while such libraries can abstract away some low-level details of message-passing code, none of the project teams adopted such libraries and they avoid writing MPI code. Even the libraries used can be problematic: On one project, the most troublesome code is the parallel HDF5 I/O library that sits atop MPI.

Other projects outside the ASC-Alliance have attempted to reuse existing class libraries to completely abstract away the low-level MPI details, but this has led to problems. For example, on one such project, some C++ code from a parallel framework wasn't portable across platforms because of differences in compiler implementations. Another obstacle to reuse is that these frameworks make assumptions about how the work will be done, and violating these assumptions requires the programmer to become involved in framework details. The effort to use the framework is roughly the same as the effort involved in writing the code using MPI, which eliminates the benefit. Because of this, each project team chose to develop a custom framework.

Both technical and sociological issues create obstacles for the adoption of a new language.

Other languages

Developers weren't aware of any alternative to MPI that could better meet their needs. Some project teams previously looked at high-performance Fortran and hybrid MPI-OpenMP as alternatives, but none felt compelled to switch.

Developers mentioned the following desirable features in an alternative parallel-programming language:

- hides memory operations from the programmer,
- provides the programmer with a single memory space,
- supports remote puts and gets,
- is easy to use,
- is efficiently implemented on all types of platforms, and
- can do everything MPI can do plus provide additional useful capabilities.

Even if a new language meets these criteria, projects wouldn't guarantee its adoption. The existing codes are already highly scalable and portable and have taken years to develop and verify. All the developers would have to be convinced that the new language would make their programs much better.

Obstacles to adoption. Both technical and sociological issues create obstacles for the adoption of a new language. Technical issues include expressiveness, performance, support for large projects, and portability. A new language would need to be expressive enough to support all of the needed algorithmic features—for example, grid- and particle-based data structures.

A parallel version of C or Fortran wouldn't be expressive enough, as the code probably wouldn't be any cleaner. The performance would have to be competitive with C. C++ is an appealing language because developers can make C++ code look like C to achieve better performance. To support large projects, the language must support separation of concerns so that developers can work on isolated pieces (some languages assume programmers have a global view of the system).

Portability. Finally, the language must be portable. The developers would have to be convinced that it could be ported to anything else in the future. If the developers have to redesign the parallelism when a new platform comes out, then the new language wouldn't even be considered. Even C++ is deemed risky because of its varying levels of support for C++ features across vendor compilers.

The main sociological issue is market share. Developers would have to be confident the new language would last well into the future. Unfortunately, this creates a chicken-and-egg problem for new languages. Some current codes are so complex that migrating to something else would be strenuous.

Even if an ideal solution were found, developers wouldn't adopt it immediately. They would have to experiment with the technology. They would also like to see several large workhorse applications converted and benchmarked. Being able to use the existing code base in some fashion would make the transition easier, although developers would still consider a new language that didn't allow this if it satisfied their other requirements.

Measures of productivity

Developers suggested several useful measures of productivity. One was scientifically useful results—sufficient simulated time and resolution, plus sufficient accuracy of physical models and algorithms—over calendar time. Another measure was problem size over calendar time, where the goal is to run a larger simulation over shorter calendar time. One developer gave the example of running a particular simulation in a calendar week instead of a calendar quarter. This is related both to code speed and machine availability.

Implementation. Other suggested measures had to do with time to implementation: Minimize the time between an algorithm's conception and its implementation on a parallel architecture or the time to perform minor or substantial code modifications. For users to be productive, they must move quickly from equations to mapping equations to a parallel model, and then to getting an implementation and running it on a machine. Even a petascale machine would be too small for some of the problems being addressed.

Developers spend most of their time trying to map the solution to what's achievable given today's computing resources, so that the problem is solvable in a reasonable period. Therefore, they focus on developing more efficient algorithms, and they're more concerned with how long it will take to develop a solution than how long a particular run takes. They have a much longer-term view on performance issues than high-end computing system vendors.

For a professor overseeing students, one suggested metric was the time between a grad student completing second-year coursework and becoming a productive researcher. This involves acquiring skills as a developer and designer of parallel algorithms, and understanding the physics and how parallelism applies to it.

Infrastructure. It would have been more productive if each project team didn't have to build its software infrastructure largely from the ground up. One developer related how, after a computational scientist at Lawrence Livermore National Laboratory designed and programmed an algorithm, the program was handed off to an applications programmer who would rewrite it to run quickly on the system: A good test of a productive environment is when the applications programmer doesn't need to rewrite the computational scientists' programs.

Some productivity issues were only tangentially related to expressing parallelism in the code. One developer brought up the importance of verifying and validating to a project's productivity, especially given the substantial learning curve involved in adopting a formal verification and validation process, which the national laboratories pushed several years ago. The ability to estimate the effort required to implement features and the resulting performance of the code outside users' code reuse, and external users, easily modifying the code to suit their own needs were among other productivity issues mentioned.

These projects face unique challenges both because of the nature of the problem and the difficulties associated with programming the environment. Because the projects we examined are based in academic environments, it's unclear how much they would generalize to computational science projects in industry or government. ■

Acknowledgments

We acknowledge Chuck Wight, Steve Parker, Anshu Dubey, Edwin van der Wedie, Frank Ham, Gianluca Iaccarino, Dan Meiron, Michael Aivazis, Mark Brandyberry, and Robert Fiedler for providing us with the information presented in this article. We also thank Robert Voigt for creating the opportunity to conduct these interviews. This research was supported in part by Department of Energy grant awards DE-FG02-04ER25633 and DE-CFC02-01ER25489, Air Force Rome Labs grant award FA8750-05-1-0100, and NASA Ames Research Center grant NNA04CD05G.

Reference

1. L. Hochstein et al., "Parallel Programmer Productivity: A Case Study of Novice Parallel Programmers," *Proc. 2005 ACM/IEEE Conf. Supercomputing* (SC 2005), ACM Press, 2005, p. 35.

***Lorin Hochstein** is an assistant professor in the Department of Computer Science and Engineering and a member of the Laboratory for Empirically Based Software Quality Research and Development at the University of Nebraska-Lincoln. His research interests are experimentation in software engineering, software technology evaluation, and software engineering for computational science. Hochstein received a PhD in computer science from the University of Maryland. He is a member of the IEEE Computer Society and the ACM. Contact him at lorin@cse.unl.edu.*

***Victor R. Basili** is a professor in the Department of Computer Science and Institute for Advanced Computer Studies at the University of Maryland, and chief scientist and executive director of the Fraunhofer Center for Experimental Software Engineering. His research interests are measuring, evaluating, and improving the software process and product. Basili received a PhD in computer science from the University of Texas at Austin. He is a Fellow of the IEEE and the ACM. Contact him at basili@cs.umd.edu.*