

High Productivity Computing Systems Program

**The ASC-Alliance projects:
A case study of large-scale parallel scientific code
development**

**Lorin Hochstein
Victor R. Basili**

**Computer Science Department
University of Maryland
College Park, Maryland 20742**

**Technical Report CS-TR-4834
UMIACS-TR-2006-50**

October 2006

Abstract

Computational scientists face many challenges when developing software that runs on large-scale parallel machines. However, their software development processes have not previously been studied in much detail by software engineering researchers. To better understand the nature of software development in this context, we examined five large-scale computational science software projects, known as the ASC-Alliance centers. We conducted interviews with project leads from all five of the centers to gain insight into the nature of the development processes of large-scale parallel code projects, and to identify issues in the current state-of-the-practice that reduce programmer productivity. The results of the interviews are summarized in this report.

1 Introduction

“Computational science” refers to the application of computers to advance scientific research through the simulation of physical phenomena where experimentation would be either prohibitively expensive or simply not possible. Advancement of scientific research depends upon the ability of these scientists to develop software productively, and is therefore of interest to software engineering researchers. However, there are many differences in software development process in this domain compared to other domains such as IT. One major difference is that scientific software can be very computationally demanding and may require the use of the most powerful machines available today, which are sometimes referred to as high-end computing (HEC) systems or supercomputers. These systems present some unique challenges to software development.

To learn more about the process of developing software to run on HEC systems, we studied five different software projects that develop such codes¹. These projects make up a group of research centers known as the ASC-Alliance centers. Each center owns a large software project which is focused on addressing a different problem in computational science. These centers are provided with access to large-scale HEC systems located at various supercomputing centers.

To study these projects, we conducted interviews with high-level members of each project. These project members were all involved in project management, software architecture, or software integration. Our goals were to characterize product, project organization, and process, both in terms of using the software and developing the software, and to identify particular challenges faced by the developers. Note that we use the terms “developer”, “scientist”, and “programmer” interchangeably in this report.

1.1 Background

The five ASC-Alliance centers were formed around 1997 by the NNSA (an agency within the Department of Energy) to develop computational simulation as a credible method of scientific research [1]. The major project at each center is focused on solving

¹ In the HEC community, scientific computer programs are referred to as “codes”

one particular scientific problem by developing multi-physics, coupled applications. This refers to the simulation of different aspects of physical phenomena (e.g. solid mechanics, fluid mechanics, combustion), which are then “coupled” together to form a single simulation. The five centers are:

- Center for Simulating the Dynamic Response of Materials, California Institute of Technology
 - Goal: simulate the response of materials to strong shocks
- Center for Integrated Turbulence Simulations, Stanford University
 - Goal: simulate full-scale jet engines
- Center for Astrophysical Thermonuclear Flashes, University of Chicago
 - Goal: simulate thermonuclear burn of compact stars
- Center for the Simulation of Advanced Rockets
 - Goal: simulate solid propellant rockets
- Center for Simulation of Accidental Fires & Explosions, University of Utah
 - Goal: simulate fires in contained vessels.

2 Goals and methodology

Our goals for conducting this study are to characterize which scientific programming activities are time-consuming and problematic, common problems that scientific programmers face, and the impact of software technologies on developer effort.

We conducted this study within the context of the DARPA High Productivity Computing Systems (HPCS) project [2]. The goal of the HPCS project is to improve the productivity of computational scientists through the development of technologies such as new machine architectures and new parallel programming languages. 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 [3]. However, without empirical data on how scientific codes are actually developed, we have no larger context for interpreting our results. In particular, we did not know whether a new parallel programming language would address the major problems that the developers faced, or whether they would adopt a new language if given the opportunity.

To conduct this study, we began by distributing a pre-interview questionnaire to each center that asked for some basic information about the project. Next, we conducted a telephone interview with one or two of the technical leads on the project. From this interview, we generated a summary document, which was sent back to the technical leads for review and corrections. Once this document had been corrected, we generated this technical report that combines the results across all of the projects. Both the questionnaire and interview guide can be found in the appendix

3 Software characteristics

While our main object of study was the software development process of the scientists, we first wanted to characterize the product that they were working on, so we had some

context for the software environment that the developers were working in. We asked about *attributes* of the software (information about the size of the codes, degree of code reuse via libraries, organizational structure of the code), and the intended *machine target* (what kinds of machines the codes are intended to run on).

3.1 Attributes

The size of the codes range from 100-500 KLOC. Most of the codes are written as a mixture of C/C++ and Fortran, with one code being 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 pre-existing codes.

All codes make use of the MPI library to achieve parallelism. In addition, each code makes use of external libraries, for features such as I/O (HDF, NetCDF, CGNS, Panda), mesh operations – including adaptive mesh refinement, (PARAMESH, Mesquite, Metis, MeshSim, SAMRAI), computational geometry (CGAL), linear algebra (BLAS, LAPACK), and tools for solving sparse linear systems and systems modeled by partial differential equations (PETSc, HyPre, CLAWPACK).

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

Each code is organized into independent subsystems, and the subsystems are maintained by separate individuals or small groups. All codes use a component-based architecture to minimize coupling between individual subsystems. In several of the projects, these independent subsystems are almost like separate projects: they can run as standalone applications and may incorporate new features that are independent of the larger, coupled application. Since almost all of the codes involve multiple programming languages, they must deal with language interoperability issues. (The one exception, a pure Fortran application, once used a Python framework to drive the application but abandoned it because of the difficulty in porting a hybrid Python/Fortran application to multiple platforms). One project built their component framework around the Common Component Architecture (CCA), which is a community effort to simplify the task of building such multi-language, 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.

3.2 Machine target

The codes are designed to run on “flat” MPI-based machines (i.e. all communication takes place through message-passing, even if some processes share physical memory). While all of the codes currently run on clusters of symmetric multiprocessors (SMPs), none of them have 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 would not 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 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 project found that a pure MPI implementation was competitive with hybrid MPI-OpenMP approach, and the other did observe increased performance when incorporating OpenMP but have not had a chance to follow up on this work due to other priorities.

4 Project organization

Like all software projects that involve more than a single individual, the developers on these projects must coordinate their efforts. We wanted to understand the *organizational structure* (how the project was organized), the *staff* (who the developers were) and their *configuration management* process (how the developers coordinated to make changes to the code). We were looking for similarities and differences with software projects in other domains, and whether the scientists encountered any domain-specific issues from a project management point of view.

4.1 Organizational structure

Each project is divided up 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 each subsystem is owned by one of the groups. 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 changes to the code. Each project also has either a chief software architect or a group who is responsible for the integration code.

Development is compartmentalized, and the groups are relatively independent. There are integrated code development meetings once a week where the core developers from each of the groups meet to discuss issues such as coordinating code changes that will affect more than one module.

4.2 Staff

In total, there are about seventy-five people actively involved on a given project. Ten to twenty-five of these people are core developers that routinely contribute code to the project. The developers consist of professionals, professional staff members with M.S. and PhD degrees, postdocs, and graduate students. Their backgrounds are in physics, chemistry, applied math, engineering (mechanical, civil, aerospace, chemical), and computer science. The experience of the programmers ranges from five to twenty-five years of sequential programming, and zero to fifteen years of parallel programming. The projects also have graduate students who work on the code as part of their research, though they are not core developers.

4.3 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 adopted a formal process for approving code before it is checked into the repository. Instead, there is general agreement that test cases should pass before commits are made to the repository. Developers are individually responsible for performing any unit testing, standalone testing, and integration testing that may be necessary. On one project, all developers are automatically notified by email whenever code is checked in to the repository so that the developers are aware of recent modifications that might affect them.

Since all codes are in active use for scientific research and active 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 that is in active use. On the other projects, defect tracking is accomplished through informal communication among project members and through the use of wikis. Some projects have attempted in the past to introduce defect tracking systems, but these systems were not adopted by the developers.

5 Software usage

Our study focused mainly on issues related to the development of the software. However, we also wanted to get a sense of how the software was used, and *who was using it*. Since problems with requirements are a major issue in other domains of software engineering, and requirements are often driven by user needs, we wanted to understand the role of the *user* in the software development process in this domain. In addition, we wanted to understand what the *execution times* were like. We did not know just how long these types of programs took to run, and we believed going into the study 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*.

5.1 Users

The main users of the codes are research scientists who are the active developers. Some of the users are students who are using the software for their own scientific research, and are not active in the code development, but these efforts are not the primary concern of the Centers. Some codes have found a user base outside of the project. These external users may even modify the program to suit their own needs.

5.2 Execution times

Characterizing the execution times of the codes is difficult because execution times vary enormously depending upon the size of the problem. Typical runs are on the order of ten to one hundred hours of execution time.

5.3 Setting up the input

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 initial conditions of the simulation. 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 the inputs is a very time-consuming task. Some of the codes simulate systems with intricate geometries (e.g. 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.

It can also take hours to weeks to retrieve the physics data needed to run the software, depending on what type of data is needed and how good the existing documentation is. In some cases, determining the correct initial conditions for the simulation is also an active area of research.

5.4 Examining the output

Users apply visualization tools for examining the output of the simulations. The projects use a mix of visualization tools developed in-house (e.g. FLASHVIEW, Rocketeer, SCIRun) and third-party tools (e.g. IDL, TecPlot, EnSight, ParaView, OpenDX, MATLAB, Iris Explorer, VisIt).

6 Development activities

The developers engage in different activities during the course of development. We asked for details about the following categories: *adding new features* to the code base, *testing* the code to verify correctness, *tuning* the code to improve performance, *debugging* the code to remove defects, and *porting* the code to new platforms.

6.1 Adding new features

Each year, the centers plan on running a major set of simulations. These simulations drive an implementation plan which determine what new features need to be added to the software. Scientists can explore avenues of research, but the overall direction is set by the implementation plan. For centers with larger external user bases, sufficiently strong demand from outside users can also drive the addition of new features.

New features added to the codes can be classified into two categories: those that are localized within an individual subsystem (low-level change), or those that involve changes across sub-systems (high-level change). Low-level changes are administered solely by the 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 fairly mature and are currently being applied to do real science. Very few new subsystems are

planned for development, although there continue to be enhancements to existing subsystems. Most modules have satisfactory parallel performance, with the exception of very new modules and modules where efficient parallelization is still an open research problem (e.g. adaptive mesh refinement).

In some of the projects, the developers do not have to write code explicitly in parallel but instead build on top of a parallel infrastructure that abstracts away the parallelization details. Other projects require that the developers program directly to the MPI library.

6.2 Testing

All projects use a suite of regression tests to catch any errors introduced by programmers modifying the code. Some projects have an automated system for running regression tests, and others run the regression tests manually. One project requires that new students who check out the code are required to run the regression tests as part of their learning process.

Testing a new algorithm is a challenging task in this environment. It is not sufficient to define simple test cases where modules are fed known inputs and checked against expected outputs. Rather, algorithms are evaluated in terms of qualities such as stability, accuracy, speed, and linear scalability. A module is considered to be functioning correctly if, for the class of inputs that are of interest, the quality of the module's output is sufficient to allow it to be coupled with other modules and produce coupled applications. Since the inputs of interests change over time as more complex simulations are attempted, an algorithm that is acceptable today may not be acceptable tomorrow. Therefore, the testing process is different from other software domains because the focus is on identifying *algorithmic defects* (i.e. evaluating the quality of the algorithm) rather than on *coding defects* (i.e. 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 the quality of algorithms involves qualitative analysis about how the algorithm behaves. There are different strategies for testing an algorithm, depending upon the nature of the problem (e.g. checking if certain quantities are exactly or approximately conserved, checking if symmetry properties hold, checking against known analytical solutions). Some of the 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 cannot diverge.

In general, the developers do not know whether an algorithm solves an equation correctly until certain requirements are passed. For example, a module may appear to be performing correctly in isolation, but when used in a coupled application it may behave in unexpected ways. This testing process is very interactive and 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 a lot of guidance from senior people who understand the broader scope of the physics and software.

6.3 Tuning

Tuning activity occurs when the developers discover that the software is executing much slower than expected. Tuning may be required when the software is being ported to a new platform (see also the *porting* section below), or if there major changes to the software architecture have caused performance penalties, or simply because of changes made to a particular subsystem create a bottleneck. At least one project uses tuning specialists: developers who are skilled at identifying and fixing performance bottlenecks. One particular tuner comes from a local computer science group that develops performance analysis tools.

Since one of the project goals is to develop algorithms that will last across many machine lifetimes, it is not seen as productive to try to maximize the performance on any one 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 a state of flux, as new algorithms are continually being evaluated, which involves changing the core components of the code. If there were many machine-specific optimizations in the code, it would be much more difficult to understand the code, which would increase maintenance effort.

For a given application, there is often a considerable amount of tuning that needs to be done in order 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 the individual projects do not focus on maximizing performance on any one system, they are occasionally able to take advantage of a team of third-party experts who can achieve a large speedup on a particular system.

Developers do use externally developed profiling tools (e.g. TAU, Jumpshot, SpeedShop, Shark). However, on some of the codes, external profiling tools have failed because they could not deal with an application written in multiple languages. In addition, some of the codes contain their own profiling routines. Some developers find these tools useful, but others say that they are familiar enough with the code that these profiling tools do not reveal any information that is not already known.

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

6.4 Debugging

All projects mentioned make some use of TotalView, which is a popular parallel debugger. Sequential debugging tools such as Purify and Ensure are also used, although they are only useful if the failure can be reproduced when running the program on a single processor. The use of trace statements to debug is common across all projects, although these are difficult to interpret when the program is running on a large number of processors. The developers also examine the simulation outputs with visualization tools to help identify defects.

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 very effectively on a moderate number of CPUs.

There are some types of defects for which the existing tools do not provide any additional assistance. Most of the debugging time is spent in algorithmic debugging (i.e. dealing with *algorithmic defects*, see the section on *testing*, above), a process for which typical debugging tools are not appropriate.

Large machines are generally batch-scheduled, which makes debugging more difficult. Since the debugging process typically involves frequent re-running of the code, running under a batch queue can become a week-long process where it would be a matter of hours on a dedicated machine where jobs could be run interactively. The National Labs make available dedicated weekends to the ASC-Alliance centers where a fraction or the entire platform of a particular machine is available for interactive use. There are a few of these weekends each year, and about 60 hours of total compute time are provided to the center, which allows them to run very large jobs and to do large-scale debugging. Debugging runs that involve smaller number of processors are done internally.

6.5 Porting

Porting activity most commonly occurs when the DOE purchases a new machine that becomes available for project use. Additionally, porting work may be necessary due to a software upgrade on a system that is already in use. The time required to port to a new system ranges from a single day to several weeks depending on the individual code and the maturity of the development tools on the new platform.

Porting requires a non-trivial amount of effort because, when a new platform is released, the code cannot simply be recompiled and run. Much of the porting effort is spent on a large number of small details that need to be addressed. For example, the build scripts (e.g. makefiles) need to be modified to accommodate the differences in development tools on the new system. Immature compilers on new systems are another source of porting effort; one project spent a year getting the code to run on one particular machine because of C++ issues; while some problems had workarounds, others required that they wait for the vendor to fix the compiler. One project completely abandoned the use of Python to reduce porting effort.

Although porting is only a small percentage of the total effort, it can involve a great deal of work in a very focused time (e.g. several people working for a month). Porting is perceived as a task that involves an unwarranted amount of effort. Some projects have broken compilers and MPI codes on every platform. Two projects did not port their code to ASCI Red because the Fortran compiler did not support needed features properly, and

it was not worth the effort to try and work around these problems to get the code to run on the machine.

6.6 Effort distribution and bottlenecks

Developers report spending most of their development time (75-95%) on adding new features and testing. Tuning and porting take up relatively little of the total development time (1-10%). However, the distribution of effort varies depending on the development phase of the project. Earlier on in the projects, when new data structures had to be designed to handle a particular class of physics problems, effort distribution was different than currently where the effort on data structures is oriented towards modifying them for new areas. The projects occasionally undergo large-scale re-architecting to better incorporate new features, and this also changes the effort distribution.

Verification and validation was reported as a common bottleneck. In particular, debugging parallel algorithms was a bottleneck across all projects. Debugging is made more challenging because of the defects that only manifest themselves in more complex execution environments where it is more difficult for the programmer to observe what is happening. For example, a code may run perfectly on 32 processors, but fail on 64 processors. Or, a subsystem may work well when executing independently, but does not 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)
- Understanding old code (when re-architecting)

7 General observations

Based on the interview responses, we made some general observations about software engineering in this domain. One surprise to us was the challenging nature of performing *validation* of this type of software. We were also very interested in the role that *MPI* played in these projects, because of the goals of the HPCS project to develop alternative parallel programming languages. In particular, we were interested in understanding how likely an alternative to MPI, either in the form of a *library or framework* that encapsulates MPI, or in the form of another *parallel programming language*. Finally, we were interested in the scientists' opinions on what *productivity* means to them, since one of our ultimate goals as software engineering researchers is to help improve programmer productivity.

7.1 Validation

Validation of the codes (checking the simulation accurately simulates the phenomena of interest) is a formidable challenge. A validation study is typically a research project or the essence of a thesis. A student will choose a problem that has an interesting set of experimental data associated and then identifies to what extent it can be simulated.

Because of the effort involved in running such studies, it is not feasible to validate every new algorithm with an experiment.

7.2 MPI

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

Despite MPI being the most appropriate technology for the job, there was 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 required by MPI 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 of the advantages of MPI is that it is possible to write MPI code knowing very few functions, so it is very easy to teach. While teaching people to use MPI is not particularly hard, teaching people to write MPI effectively is extremely difficult: this fact distinguishes the first year graduate student from the developers who have been at a center for four-to-five years.

7.3 Alternative to MPI: libraries/frameworks

All of the projects used libraries that were built on top of MPI. However, while such libraries can abstract away some of the low-level details of message-passing code, none of the projects adopted such libraries to the extent that they could entirely avoid writing MPI code. Even the libraries that are 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 of 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 was not 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 that the programmer become deeply 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 of use. Because of this, each project chose to develop a custom framework.

7.4 Alternative to MPI: other languages

There was no alternative to MPI that any of the developers were aware of that could better meet their current needs. Some projects had previously looked at High-

Performance Fortran and hybrid MPI-OpenMP as alternative models, 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
- Can do everything MPI can do plus provide additional useful capabilities

Even if a new language met the above criteria, this would not guarantee adoption by the projects. The existing codes are already highly scalable and portable, and have taken years to develop and verify. All of the developers would have to be convinced that the new language would make their programs much better.

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 (e.g. grid-based data structures, particle-based data structures). A parallel version of C or Fortran would not be considered expressive enough, as the code would probably be no cleaner than it is currently. The *performance* would have to be competitive with C. C++ is an appealing language because C++ code can be made to 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 the programmer has a global view of the system). Finally, it must be *portable*. The developers would have to be convinced that it could be ported to anything else in the future. If the developers had to re-design the parallelism when a new platform comes out, then it won't even be considered. Even C++ is considered a somewhat risky technology because of the varying levels of support for C++ features across vendor compilers.

The main sociological issue is that of *market share*. The developers would have to be confident that the new language would last well into the future. Unfortunately, this creates a chicken-and-egg problem for new languages. Some of the current codes are so complex that it would be very strenuous to migrate to something else. Even if an ideal solution were found, it would not be adopted immediately. The developers would have to experiment with the technology first. They would also like to see a number of large workhorse applications converted and benchmarked. Being able to use the existing code base in some fashion would make the transition easier, although they would still consider a new language that did not allow this, if it satisfied their other requirements.

7.5 Measures of productivity

The developers suggested several measures of productivity that would be useful in their context. One was *scientifically useful results* over calendar time, where *scientifically*

useful results implies sufficient simulated time and resolution, plus sufficient accuracy of the physical models and algorithms. Another was *problem size* over calendar time, where the goal is to run a larger simulation in a smaller calendar time. One developer gave the example of running a particular simulation in a calendar week instead of a calendar quarter. This is partially related to the speed of the code, and partially to the availability of the machine.

Other suggested measures had to do with time to implementation: minimize the time between conception of an algorithm and the actual implementation on a parallel architecture, or the time to make modifications to the code (minor or substantial). For users to be productive, the holy grail is to very quickly go from equations to mapping the equations to a parallel model, and then getting an implementation and running it on a machine. Even a petascale machine would be too small for some of the problems that are being addressed. Most of their time is spent trying to map the solution to what is achievable given today's computing resources, so that the problem is solvable in a reasonable period of time. Therefore, they are focused on the development of more efficient algorithms. They are more concerned with how long it will take to develop a solution than how long a particular run takes. The projects have a much longer-term view on performance issues than HEC vendors.

For the professor whose job it is to turn out students, one suggested metric was the length of time from when a grad student finishes the second year of coursework to when she is a productive researcher in the group. This involves acquiring skills as a developer, as a designer of parallel algorithms, understanding the physics and how parallelism applies to it. At the university, there are considerably more educational aspects to the notion of productivity than in a government or industrial lab.

Each project had to build up a software infrastructure, largely from the ground up. This required a substantial amount of effort, and it would be more productive not to have to build such infrastructure. One developer related how at Livermore Labs, after a computational scientist would design and program an algorithm, the program would be handed off to an applications programmer who would then rewrite it so it would run quickly on the system. One developer noted that a good test of a productive environment is when, in such an environment, the applications programmer does not feel the need to rewrite the computational scientists' programs.

Some productivity issues were only tangentially related to the details of expressing parallelism in the code. One developer brought up the importance of doing verification and validation to the productivity of the project, especially given the substantial learning curve involved in adopting a formal verification and validation process, which was pushed by the National Laboratories several years ago. Other productivity issues mentioned were the ability to estimate the effort required to implement features and the resultant performance of the code, the degree of code reuse by outside users, and the ease of which external users can modify the code to suit their own needs.

8 Conclusions

In this paper, we have attempted to capture different aspects of the software development process for large-scale HEC applications, focusing on areas that are particularly challenging. These projects face some unique challenges, both because of the nature of the problem, and because of the difficulties associated with programming the environment. Note that because all the projects that we examined are based in academic environments, it is unclear how much would generalize to computational science projects in industry or government.

9 Acknowledgments

We would like to 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 contained in this report. We would also like to thank Robert Voigt for creating the opportunity to conduct these interviews.

10 References

- [1] Douglass Post, personal communication.
- [2] <http://www.highproductivity.org>
- [3] Lorin Hochstein, Jeff Carver, Forrest Shull, Sima Asgari, Victor R. Basili, Jeffrey K. Hollingsworth, Marvin V. Zelkowitz. "Parallel Programmer Productivity: A Case Study of Novice Parallel Programmers." *International Conference for High Performance Computing, Networking and Storage (SC'05)*. November 2005.

Appendix A HPC Project Questionnaire

1. General

What type of problem is your code project trying to solve?

2. Activities

Where is most of the team's time spent when *developing* the software?

- Adding new features (e.g. new algorithms to improve accuracy)
- Fixing bugs
- Improving performance
- Porting to new platforms
- Other: _____

Where is most of the team's time spent when *using* the software?

- Setting up the input (e.g. setting configuration options)
- Executing the code
- Analyzing the output (e.g. visualization)
- Other: _____

How long is a typical run of the program? (e.g. 20 hours)

3. Hardware

What machines do you run your software on? (e.g. 64-node Linux cluster, 128-processor SGI Altix, ASCI Red)

4. Software

What programming languages are used? (e.g. Fortran, C, C++, Java, Python)

What is the underlying parallel technology?
(e.g. MPI, OpenMP, threads, PVM, HPF, Co-Array Fortran)

What libraries or frameworks are used? (e.g. ScaLAPACK, PETSc, POOMA)

What type of parallelism/communication patterns are involved in the software?
(e.g. nearest-neighbor, wavefront, all-to-all, embarrassingly parallel)

5. Human

How many people are actively involved with the project? _____

What are their academic backgrounds? (e.g. professor of chemical engineering, PhD student in CS, professional programmer with MA in EE)

How many years of experience do they have in programming?

How many years of experience do they have in *parallel* programming?

6. Productivity

What software-related issues do you encounter that reduce your productivity?
(e.g. tasks that consume more programmer time than they should, programs that seem too difficult to learn/use)

How could we as empirical researchers help you?
(e.g. help justify a tool purchase by demonstrating how much time is currently spent in certain tasks without the use of that tool)

Would you be willing to follow-up this questionnaire with an interview?

No

Yes Please give contact information _____

Appendix B Interview Questions

1 Product

1.1 Attributes

Do you have a name for the software?

How large is the entire codebase, in terms of lines of code (excluding external libraries)?

How many major subsystems are there? (Is the program one monolithic application or are there subsystems that can be used in isolation?)

Roughly speaking, what percentage of the code is:

- Custom (in-house)
- Libraries developed and maintained in-house, but also used by external projects
- Libraries maintained externally

Did any of the external libraries have to be modified at all for use on the existing project?

Did you build the architectural framework for connecting the different subsystems from scratch, or did you build upon another framework (e.g. Common Component Architecture)?

1.2 Machine target

Is the code optimized for a particular machine or class of machines? (e.g. cluster of SMPs?)

Do you measuring the scalability of the code (e.g. weak scaling, strong scaling)? How well does the code scale?

1.3 History

What is the history of the codebase? Was it all written from scratch at the beginning of the project, or are some subsystems reused from prior projects?

2 Project organization

2.1 Organizational structure

How is the development team structured? Are developers divided up into groups? If so, how do they coordinate changes?

2.2 Staff

How many of the project members are core developers?

2.3 Configuration management

Do you use version control? How do you handle the issue that some users require a stable version of the code, while developers are actively modifying the code? What are the rules for when code can be checked in to the repository?

Do you use a bug tracking system? If not, how do you communicate information about bugs to other project members?

What kinds of documentation exist for the software? (e.g. user guides, design docs, etc.)? How often are these updated?

3 Using the software

3.1 Users

Who are the main users of the software? If it is primarily the developers, are there any users who are not developers?

3.2 Setting up the input

How do you set up the input for the program?

- GUI interface for setting up input
- input files
- other

How much time does it take to set up the input for a run?

3.3 Examining the output

What do you use to visualize results? Are the visualization tools developed in-house, or are they third-party tools?

4 Development activities

4.1 Adding new features

How do you decide what additions/modifications are made to the code that only affect one subsystem?

How do you decide what additions/modifications are made to the code that affect the global behavior of the system?

Can you walk us through the stages of modifying the program from the initial step of deciding to add a new feature to the final step of the new code being accepted into the code-base?

Do you plan on adding any new subsystems?

Do the developers today program directly to MPI or do they program to an interface built on top of MPI? If it is an interface on top of MPI, how do they express parallelism?

4.2 Testing

How do you do testing/V&V? Do you do regression testing?

How do you evaluate new algorithms?

4.3 Tuning

When do you tune?

Which developers are usually involved in the tuning process?

Can you describe the tuning process you use?

Do you use profiling tools? If so, which ones?

4.4 Debugging

Which strategies do you use for debugging?

- trace statements (e.g. printf's)
- serial debuggers
- parallel debuggers on large runs (e.g. hundreds of procs)
- parallel debuggers on smaller runs (e.g. ~10 procs)
- other?

4.5 Porting

How often do you port the code to a new platform?

How long does this typically take?

Where is most of the porting time spent?

Are there any machines that you considered porting to but decided against it because it turned out to be too difficult?

4.6 Effort distribution and bottlenecks

How does the development effort break down in terms of:

- better algorithms (better accuracy, speed)
- adding new features
- tuning on existing platform
- porting to new platform
- other

Where do you spend the most time?

What is the most difficult?

Where are the bottlenecks, if any?

- generating input
- adding new features
- debugging code
- testing/validation
- production runs
- other

4.7 Achieving performance

Which is more difficult:

- identifying parallelism (designing and implementing a correct parallel algorithm)
- achieving performance (modifying a “naïve” parallel algorithm to achieve reasonable performance on a given machine)

5 Programming models and productivity

5.1 Choice of parallel programming model

Given the choice, which programming model or language will you choose to work with? Under what conditions?

Why did the developers choose the particular technologies that are being used on this project?

5.2 Adopting a new language

Are there any circumstances under which you would switch from MPI to a different parallel programming language?

What are the obstacles that would keep you from using a new parallel programming language in your software?

What would a new parallel programming language have to offer in order to be worth adopting?

5.3 Productivity measures

What is a meaningful unit of “productivity” to you?

6 Follow-up

Is there some subset of the development we could study in real time? Would you be willing to let us instrument the computers or let us collect data on forms to get some ideas about where you spend your time, what kind of defects you make, or the effects of your programming model on your development effort/achieved performance? Would you be willing to have a researcher do an observational study on some programmers?