# Software Development Environments for Scientific and Engineering Software: A Series of Case Studies

Jeffrey C. Carver[1], Richard P. Kendall[2], Susan E. Squires[3], Douglass E. Post[4]

[1]*Department of Computer Science and Engineering*
*Mississippi State University*
carver@cse.msstate.edu

[2]*Software Engineering Institute – Carnegie Melon University*
rkendall@sei.cmu.edu

[3]*Sun Microsystems*
Susan.Squires@Sun.com

[4]*Department of Defense High Performance Computing Modernization Office*
post@hpcmo.hpc.mil

## Abstract

*The need for high performance computing applications for computational science and engineering projects is growing rapidly, yet there have been few detailed studies of the software engineering process used for these applications. The DARPA High Productivity Computing Systems Program has sponsored a series of case studies of representative computational science and engineering projects to identify the steps involved in developing such applications (i.e. the life cycle, the workflows, technical challenges, and organizational challenges). Secondary goals were to characterize tool usage and identify enhancements that would increase the programmers' productivity. Finally, these studies were designed to develop a set of lessons learned that can be transferred to the general computational science and engineering community to improve the software engineering process used for their applications. Nine lessons learned from five representative projects are presented, along with their software engineering implications, to provide insight into the software development environments in this domain.*

## 1. Introduction

Software written to solve scientific and engineering problems that require the use of high performance (massively parallel) supercomputers is an increasingly important class of applications. Software engineering processes and methods used to develop these applications varies from those used to develop other types of applications, such as in commercial IT. Some of these differences are highlighted in Section 2.

The main goal of the work described in this paper is to better understand the software engineering issues that specifically influence the success or failure of scientific and engineering software. This paper describes a series of case studies of five such projects, sponsored by various US government agencies.

The remainder of the paper is organized as follows. Section 2 provides the motivation for studying scientific and engineering projects. Section 3 gives an overview of the case study methodology used to gather the data. The five projects studied are described in Section 4. Section 5 discusses the lessons learned drawn from the whole set of studies along with their implications for software engineering. Finally, the conclusions are presented in Section 6.

## 2. Background and Motivation

Application software developed specifically for scientific and engineering purposes (often using high performance computers) has not been extensively reported on in the software engineering literature. This type of software can be developed for: simulations of physical phenomena, processing of large amounts of data, performance of complex calculations, and many other important problems that could not be solved by the development of traditional software. The presence of some unique characteristics of this type of software makes its study both interesting and valuable.

The first unique characteristic that differentiates scientific and engineering software from its commercial IT counterpart is the requirements

29th International Conference on Software Engineering (ICSE'07)
0-7695-2828-7/07 $20.00 

discovery and gathering process. While most scientific and engineering projects are ultimately based on the underlying laws of nature, which are fixed, the application of those laws to a specific problem is often unknown at the start of the project. Most requirements, beyond some obvious high-level ones, are discovered during the course of the project. Therefore, many of these projects begin as research projects without a definitive relationship among deliverables, schedule and resources.

Another distinguishing characteristic is that the main driver for these projects is, not surprisingly, correct science or engineering, rather than ensuring software quality through the use of sound software engineering practices. In fact, many of the projects are not given adequate funding or support to implement even basic software engineering principles. Another source of the often-encountered absence of software engineering discipline is that the teams are largely staffed by domain scientists and engineers rather than by formally trained software engineers.

The third unique characteristic is that the developers tend to be averse to the "process"-oriented software development approaches which have been successfully used to manage risk on other types of software projects. This aversion is due in part to the lack of formal software engineering training and in part to the nature of the application domain. Many of the projects have long life-cycles that last for decades during which the model of science embodied in the software evolves as knowledge evolves. Developers tend to think they will have greater flexibility by not following rigid software development processes.

The study of the application of software engineering principles to scientific and engineering computing has been increasing in importance and prominence. One indication of this prominence is the DARPA High Productivity Computing Systems Project (HPCS)[1]. The overall goal of the HPCS project is to develop the next generation of viable supercomputers and ensure their programmability. As part of the HPCS project, much effort has been devoted to studying the software engineering process for scientific and engineering projects. In addition, two ICSE workshops have focused on the interaction of software engineering and high performance computing [3, 4]. Papers on software engineering for scientific and engineering computing have also recently appeared in both software engineering conferences [13] high performance computing conferences [2] and other venues [10].

[1] http://www.highproductivity.org

## 3. Case Study Methodology

To gather information about software engineering within the scientific and engineering computing domain, we conducted a series of retrospective case studies. The case study approach, as defined by Yin, is a well-understood method for gathering initial information [15]. This approach employs a set of qualitative, open-ended methods to explore a topic and develop hypotheses. The methods include data collection techniques like Document Reviews, Collection of Contextual Artifacts, Self-Reporting, and Interviews. Data collected during a case study using this broad set of methods can provide foundational knowledge that is useful for generating hypotheses. Similar case study approaches have been previously been employed in software engineering [1, 8, 9, 12].

Each case study reported in this paper explored the work of one project team along with the context in which that work was performed. The overall goal for each case study was to identify, as accurately as possible, the activities and practices that contributed to the success of a project and the activities that were counterproductive. The use of a standard set of methods across the series of case studies allowed for the collection of consistent and comparable data. By conducting multiple case studies on projects from different application domains (explained further in Section 4), general lessons learned were. The data collection methods were selected based upon time, security, and the nature of the topics to be explored.

To ensure that the case studies were as successful and accurate as possible, the case study team consists of experts with different, relevant backgrounds: Computational Science and Engineering, Cultural Anthropology, and Software Engineering. In addition to their technical expertise, the case study team members also have extensive experience in the collection and analysis of survey and interview data.

The case study methodology included these steps:

1. **Identify a feasible project and its sponsor.** Gaining the support of the project sponsor (i.e. funding agency) is key for a successful case study. The sponsor's support makes the project team more comfortable participating in the case study.
2. **Negotiate case study participation with the project team and sponsor**. This negotiation involves both logistics (i.e., when and where to conduct the case study) and confidentiality (i.e., teams and individual team members must be confident that they cannot be identified by either their management or by outside readers. Also, teams must be assured that they will be given the

opportunity to review any reports to remove objectionable material prior to publication).

3. **Survey the project team with a pre-interview questionnaire.** The pre-interview questionnaire serves two purposes: 1) it reduces the amount of time the case study team has to spend conducting an on-site interview, and 2) it allows the project team members to look up information that may not be available during a face-to-face interview.
4. **Analyze the questionnaire and plan on-site interviews**. The responses to the questionnaire determine which topics need clarification during the on-site interview. Based on this information, an interview guide is created with the topics to be addressed during the interview.
5. **Conduct on-site interview.** The interview consists of three steps. First, the project team gives an overview presentation. This presentation often addresses many of the items on the interview guide and allows the project team to communicate what they believe to be most important about the project. Second, questions are posed to the entire project team. Third, if the project team is large enough, it is divided into smaller groups for focused questions. During each of these interview sessions, there is a lead questioner, who is responsible for ensuring that all questions are covered. The other members of the case study team act as scribes and also ask follow-up questions based on their expertise. When possible, the interviews are taped and transcribed for further analysis.
6. **Analyze the on-site interview and integrate with the questionnaire.** An initial list of findings is prepared by consolidating the information gathered through these two sources.
7. **Conduct follow-up with the project team to resolve unanswered questions.** The initial findings along with any remaining unanswered questions are iterated with the project team until everyone is satisfied.
8. **Draft the case study report and iterate with project team and sponsor.** Based on the information collected and the follow-up, a preliminary report is drafted. This report is circulated to both the project team and the sponsor to ensure that all parties involved agree with the accuracy and the anonymity of the report.
9. **Publish the case study report**.

The pre-interview questionnaire and on-site interviews specifically focused on the following topics:

1. Goals, requirements and deliverables
2. Project Characteristics, Structure, Organization and Identified Risks
3. Code Characteristics and Structure
4. Staffing
5. Workflow and Workflow Management
6. Verification, Validation and Testing
7. Measurement of Success
8. Lessons Learned

## 4. Description of Projects

Due to the classified nature of some of the projects, and the desire for anonymity by all of the projects, each has been given a pseudonym to remove any identifying characteristics. This approach is common when conducting case studies [15]. The following subsections provide the basic characteristics of the five projects studied, with more details on four of them published elsewhere (FALCON [11], HAWK [5], CONDOR [6], EAGLE [7], and NENE). Each project is characterized by its purpose, its development and execution environment (including languages used), and its users; this information is summarized in Table 1.

### 4.1 FALCON Project

The goal of the FALCON project, which is just beginning its useful life in its eleventh year, is to develop a predictive capability for a product whose performance involves the trade-off among many strongly coupled physical effects spanning at least ten orders of magnitude in each of the temporal and spatial scales. An accurate predictive capability is needed to reduce the dependence of the sponsor on large, expensive and potentially dangerous empirical tests to certify the product.

The parallel programming model used in this project is Message Passing Interface (MPI). The target platforms are a shared-memory LINUX cluster with ~1000 nodes and a large vendor-specific shared-memory cluster with ~ 2000 nodes. The bulk of the code is written in an object-oriented instantiation of FORTRAN. The team has successfully captured some of the advantages of object-orientation, such as polymorphism and inheritance, while avoiding the pitfalls (especially for parallel machines) of many levels of inheritance and excessive use of templating. The major blocks of code are about 410,000 SLOC of FORTRAN, 50,000 SLOC of C, 200,000 SLOC of library code, and about 30,000 SLOC of Perl, Python and Unix scripts. Perl and Python are primarily used for build and test scripts.

FALCON is used by an external team of highly knowledgeable and experienced product engineers to assess the potential behavior of new and existing product designs. Validation by the users is done by comparing the output of the software to data from past experiments and a few new experiments. The engineers' level of experience and expertise is sufficiently high that they can not only identify defects and model deficiencies, but can often identify the source of the defects and needed model improvements. In a very real sense, the users participate constructively and effectively in the development, verification and validation of the code.

FALCON is extensively documented on an internal web-site (approximately 400 Mbytes of HTML files). The documentation consists of descriptions of the physics; the algorithms and models; the input and output; and instructions for executing the code. This documentation has proved to be highly useful.

### 4.2 HAWK Project

The purpose of the HAWK project was to develop a computational predictive capability to analyze the manufacturing process of a family of composite material products. It was designed to allow the sponsor to minimize the use of time-consuming, expensive prototypes for ensuring efficient product fabrication. The manufacturing process is governed by three physical processes: 1) chemical reactions, 2) heat transfer, and 3) fluid flow through a porous medium.

HAWK employs an unstructured, fixed finite element mesh to represent and resolve the objects to be manufactured. These objects can exhibit very complex geometries which may require a significant effort to represent in HAWK (months of staff-time). The current version of HAWK evolved from an earlier version and uses a message passing architecture based on MPI (targeted to machines like the SGI Origin 3900). Portability is important for HAWK. Up to this point, it has been successfully ported to hardware developed by SGI (Origin® 3900), Linux Networx (Evolocity® Cluster), IBM (P-Series® 690 SP) and Intel-based Windows platforms.

Like FALCON, HAWK was developed using multiple languages. There are approximately 134,000 lines of executable code in the program library of which 67% are written in C++, 18% in C, and the remaining 15% primarily in FORTRAN90 and Python. The finite element "objects" and object manipulation is coded in C++ (a strong contrast to the FALCON project, which is based primarily on an object-oriented instantiation of FORTRAN77); the FORTRAN90 code comes primarily from third-party suppliers.

HAWK has been deployed to both internal and external product engineers. There are only a small number of users (on the order of tens of users), so the development team is able to provide the needed support. Work on this project is currently suspended.

### 4.3 CONDOR Project

The purpose of the CONDOR project was to develop a simulation capability to analyze the behavior of a family of materials when placed under extreme stress. The CONDOR project began in the late '80s; its earliest antecedent traces back to the late '60s. The objects of the simulations are multi-material physical entities with complex geometries. CONDOR simulations allow the sponsor to minimize the use of time-consuming, expensive tests to forecast performance, and provide an alternative to infeasible physical testing. The predictive capability addressed by CONDOR is accomplished by integrating a set of initial value equations for the conservation of: 1) mass, 2) momentum, and 3) energy.

CONDOR is supported on platforms ranging from PCs to parallel supercomputers. A typical PC application is $10^6$ cells running for a few hours to a few days. The largest application is on the order of $5x10^9$ cells executing on 4000 processors. A typical parallel application is on the order of $10^8$ cells executing on 100 to a few 100s of processors. CONDOR tends to be one of the first applications ported to new hardware platforms at its parent institution. Like FALCON, CONDOR uses MPI for parallelization and was developed primarily in FORTRAN77 (approximately 85% of the ~200,000 SLOC), with the remainder written in FORTRAN90, C, or Slang.

CONDOR has been deployed to both internal and external users. There are several thousand occasional users and hundreds of routine users. For PCs, it is distributed on a per seat basis using a home grown licensing manager similar to Flexlm®.

### 4.4 EAGLE Project

The EAGLE project had two important goals (1) to determine if parallel, real-time processing of sensor data guided by non-traditional algorithms was feasible for a particular application, and (2) to demonstrate the feasibility on specialized HPC hardware actually deployed in the field (this project is an example of an embedded HPC application). Thus, the EAGLE project was a demonstration project with a large research component and was not expected to result in a

production-ready software application. Consequently, there are currently no users.

EAGLE is the second half of a two-part project. Prior to its start, serial Matlab® prototypes were written by algorithm specialists (typically Ph.D. physicists). One of these prototypes was chosen to be a "specification" and "reference code" for the EAGLE implementation. Thus, the principal task of EAGLE was to create a parallelized C++ implementation of the algorithm and demonstrate real-time (or near real-time) performance. The use of a Matlab prototype is an important difference between this project and the FALCON, HAWK, and CONDOR projects.

The algorithmic foundations of EAGLE are based on Fast Fourier Transforms, median filtering, sorting, hypothesis testing, and direct solution methods for systems of simultaneous equations. Parallelization in MPI was straightforward due to the nature of the algorithms. A unique characteristic of EAGLE was its real-time aspect. The more common type of HPC problem involves computations on a very large data set producing a single result, whereas EAGLE receives a continuous stream of input data and produces a continuous stream of output data.

Like HAWK, EAGLE made extensive use of C++. Unlike HAWK, the EAGLE team developed the bulk of the code using an institutionally-supported C++ library (~70% or 25,000 SLOC). The use of the library and a Matlab prototype are distinctive features of this project, and are considered best practice by the host institution. JAVA is also used for pre-processing.

The target hardware for the real-time demonstration of EAGLE was a specialized computer that can be deployed on a military platform, while most of the development was performed on SUN Sparcs (Solaris®) and PC (Linux) with friendlier development environments. This platform change represents a novel, but critical, role for portability.

### 4.5 NENE Project

The NENE project is a suite of software applications that can be combined to allow researchers to calculate the properties of molecules using a variety of computational quantum mechanical models. This project began in the late 1980's and is still very active. The inherently global nature of the particle (atom, ion, electron, molecule) interactions described by quantum mechanics makes the likelihood of widespread parallelism remote. The NENE project began as a research code and has maintained a strong research flavor throughout its existence. For the most part, the funding has emphasized the domain science rather than software engineering or computer science attributes.

One important characteristic of this project is the large number of developers. Much of the development is done in a university environment; as a result, as students graduate, there is turn-over in the development team. In addition, a sizeable portion of the code is developed by external collaborators. The NENE project has a designated code librarian who ensures that any code included in a release is thoroughly tested prior to inclusion. Partially as a result of this situation, NENE has dealt with some daunting project risks with a rather unorthodox approach. For example, rather than deploying a sophisticated configuration management tool to manage the large program library, NENE has adopted the approach of hand integrating every line of code by the co-PI. This approach works because (1) the code is very modular with a slender program backbone, and (2) the distributed development community is very familiar with the code and thoroughly tests enhancements before submission (thereby making the job of integration simpler).

NENE now has about 20,000 installations and an estimated 100,000 users worldwide. There are over 5400 citations to the basic paper describing the project. Due to its widespread usage, there is a strong need for portability. There is essentially one version of the code (750,000 LOC), written in FORTRAN77 subset of FORTRAN90, that executes on all commonly used platforms (except on Windows-based PCs). A separate version of the code optimized for Windows PCs exists, but it is not supported by the core NENE team.

## 5. Lessons Learned

The analysis of the five projects described in Section 4 revealed some common patterns that resulted in nine lessons learned. For each lesson, this section presents support from the case studies and the implications of that lesson for software engineering.

### 5.1 Verification and Validation is very difficult in this environment.

Verification is the demonstration that the application correctly solves the equations embodied in the solution algorithm. Validation is the demonstration that the application accurately models all the important effects. Validation ensures that the software correctly captures the laws of nature by comparing its predictions to experimental data.

Validation is problematic because it is often difficult, or even impossible, to establish the correct output or result *a priori*. The goal of many of these

**Table 1** – Code Characteristics

| | FALCON | HAWK | CONDOR | EAGLE | NENE |
|---|---|---|---|---|---|
| **Application Domain** | Product Performance | Manufacturing | Product Performance | Signal Processing | Process Modeling |
| **Duration** | ~ 10 years | ~ 6 years | ~ 20 years | ~ 3 years | ~ 25 years |
| **# of Releases** | 9 (production) | 1 | 7 | 1 | ? |
| **Staffing** | 15 FTEs | 3 FTEs | 3-5 FTEs | 3 FTEs | ~10 FTEs (100's of contributors) |
| **Customers** | < 50 | 10s | 100s | None | ~ 100,000 |
| **Code Size** | ~ 405,000 LOC | ~ 134,000 LOC | ~200,000 LOC | < 100,000 LOC | 750,000 LOC |
| **Primary Languages** | F77 (24%), C (12%) | C++ (67%), C (18%) | F77 (85%) | C++, Matlab | F77 (95%) |
| **Other Languages** | F90, Python, Perl, ksh/csh/sh | Python, F90 | F90, C, Slang | Java Libraries | C |
| **Target Hardware** | Parallel Supercomputer | Parallel Supercomputer | PCs to Parallel Supercomputer | Embedded Hardware | PCs to Parallel Supercomputer |
| **Status** | Production | Production Ready | Production | Demonstration Code | Production |

projects is to simulate some physical phenomena. In many cases, the simulation falls into one of two categories. One type of simulation explores new science, which by definition does not have a known result. The second type of simulation cannot be experimentally replicated due to safety, expense, or legal constraints. In this case, there is no experimental result available against which to judge the accuracy of the simulation result.

In cases where a correct answer is known and available for verification, incorrect outputs or results can be determined, at which time the problem shifts to identification of the source of the problem, which can be an equally daunting challenge. The development process allows for at least three potential sources of defects. The first step in the development process is for a domain expert to create a model of nature (usually done mathematically). Defects could enter at this step if the domain expert builds an incorrect model, i.e. gets the science wrong. The second step is to translate that model into an algorithm, or set of algorithms, that can be later implemented in software. Even if the original model is correct, defects can enter at this step if the model is incorrectly encoded into an algorithm. Finally, the algorithms are implemented in software. Again, defects can enter during this step through inaccurate translation of the algorithm into code.

These issues combine to make the task of verification and validation for scientific and engineering applications very difficult. A member of the EAGLE team provided another reason why verification and validation is difficult:

> *V&V is very hard because it is hard to come up with good test cases*.

The inability to fully validate their results led the CONDOR team leader to take an approach summed up in the following comment:

> *I have tried to position CONDOR to the place where it is kind of like your trusty calculator – it is an easy tool to use. Unlike your calculator, it is only 90% accurate … you have to understand that the answer you are going to get is going to have a certain level of uncertainty in it. The neat thing about it is that it is easy to get an answer in the general sense <to a very difficult problem>."*

The implication of this lesson is that the traditional methods of testing software and comparing the output to an expected result are not sufficient. Scientific and engineering developers need to identify additional methods to ensure software quality and to describe the limits of the applicability of the software.

### 5.2 The primary language of a project typically does not change over time.

Two interesting questions arise about this community: (1) Why is FORTRAN still the dominant language? And, (2) Why have more modern, higher-level languages not been adopted? The developers of these projects have all chosen what they believed to be the best programming languages available to them. Because of the long duration of the project lifecycle,

this choice is often strongly influenced by issues like portability and maintainability. The case studies revealed that once that language is chosen, it typically does not change.

For example, developers of the NENE project chose FORTRAN because of its ease of learning for the scientists, compared with C++, (students and researchers can become productive in a few weeks) and its portability (a main goal of the project). FORTRAN also produces code that performs well and is supported on large-scale supercomputers. The NENE project has stayed with this choice even though FORTRAN is, for the most part, no longer taught in the universities. There would have to be a very compelling reason for NENE to change languages. Any new language would have to provide some added benefits without removing the advantages of FORTRAN.

This trend was observed in all of three of the long-lived projects. FALCON, CONDOR and NENE all stuck with their original choice of FORTRAN77, for the most part, despite the advances (and advantages for parallel processing environments) of FORTRAN90 and Co-Array FORTRAN.

The constancy of the programming language is also motivated by the users of the code. Many of the long-lived projects also have many users (sometimes numbering into the thousands) that interact with the software at the code level. Changing from the primary language would require retraining of these users. For most projects, the infeasibility of retraining users is another barrier to changing languages.

### 5.3 The use of higher-level languages is low.

The teams studied have avoided using higher-level languages (e.g. Matlab) for the main application code. This avoidance is due in part to the current limitations of Matlab. While Matlab is a good language for prototyping algorithms, code written in Matlab is usually an order of magnitude slower than code written in C, C++, or FORTRAN. Thus, Matlab is generally used to develop and test solution algorithms. Once a successful algorithm is developed, it is then recoded in another language to achieve improved performance.

Specifically, CONDOR used 85% FORTRAN77 with the remainder in FORTRAN90, C, or Slang. HAWK contains 67% C++ and 18% C, with the majority of the other 15% being FORTRAN and Python. FALCON is mostly in FORTRAN, but it also contains code written in at least 5 lower level languages (e.g. Python and Perl). Unlike the first two projects, the FALCON project did introduce object-oriented features in a backplane with FORTRAN 77/90 modules, but performance constraints greatly restricted their use. EAGLE was the only project that used Matlab. But, even in this case, it was limited to implementation of prototypes and development of the specification for the executable code, most of which was ultimately written in C++.

While C++ is surely a higher level language than FORTRAN, an interesting fact is that HAWK and EAGLE restricted their use of C++ to a set of features that fell mostly within the C subset of C++ (i.e. the higher-level features of C++ were avoided).

A specific example of the motivation for avoiding higher level languages can be seen in a comment made by one of the CONDOR developers (addressing the efficiency of the code created by a compiler):

> *I'd rather be closer to machine language than more abstract. I know even when I give very simple instructions to the compiler, it doesn't necessarily give me machine code that corresponds to that set of instructions. If this happens with a simple do-loop in FORTRAN, what happens with a monster object-oriented thing?*

The implication of this lesson along with the previous lesson is that scientific and engineering developers place more constraints on the choice of programming language than developers in the commercial IT domain. To be adopted by scientific and engineering programmers, a language has to be easy to learn, offer reasonably high performance, exhibit stability, and give developers confidence in the validity of the resulting machine instructions.

### 5.4 Developers prefer the flexibility of the UNIX command line over an IDE.

The case studies revealed an absence of IDE (Integrated Development Environments) usage by the developers. The experienced developers tended to dislike the rigidity they felt most IDEs imposed on their development activities. These developers typically knew what they wanted to do and were much more comfortable, and they believed, more efficient when typing commands on the command line rather than navigating a series of nested menus. The reasons that IDEs tend to be avoided is summed up by a member of the EAGLE team. The developer believed that IDE's were not helpful because:

> *They all [the IDEs] try to impose a particular style of development on me and I am forced into a particular mode.*

Another common belief was that developers had more control when they were "closer to the metal" (i.e. interacting more directly with the hardware). The implications of this lesson are that developers do not adopt IDEs because 1) they do not trust an IDE to automatically perform a task in the same way they would do it themselves; and 2) they expect greater flexibility than is provided in current IDEs; 3) they may prefer to use what they know rather than change.

### 5.5 Externally developed software is a risk.

Because of the very long development and deployment phases, these projects tend to avoid using externally developed software that may disappear or become unsupported during the lifetime of the project. Rather than relying on such software, the teams prefer to develop this software in-house or to use open-source software. Open-source software provides the benefit of decreased development time through the use of externally developed software while at the same time providing the development team with the security that the software will not disappear (because the team has access to the source code and could maintain it if other support became unavailable).

One exception to this rule is the NENE project, which made use of a large amount of externally developed software. In order to address the inherent risk in this approach, a librarian was designated to thoroughly test any code before integrating it into the main codebase. The developers set the project up so that the presence or absence of any externally contributed software does not endanger commitments to the sponsor. Therefore, NENE is not vulnerable to external developers failing to deliver. It is vulnerable to the core developers failing to deliver.

Ironically, the main problem faced by many computational science and engineering projects is the lack of good tools for parallel development (especially for debugging). When quality commercial tools are developed and become successful, the host company is often bought by another company and the tool is discontinued. If a project has planned its development activities around the presence of such a tool, they must then scramble to find, or more likely, develop an adequate replacement.

The implication of this lesson is that the tool situation in HPC development is tenuous because of the tension between code developers and tool developers. On one side, the code developers do not trust the longevity of or support for third-party tools. On the other side, the tool vendors cannot spend the effort required to build and support proper tools due to the lack of a stable market. Therefore, this Catch-22 situation has resulted in low usage of external software and tools. The problem of third-party software has also been identified by other researchers [14].

### 5.6 Performance competes with other important goals

While performance (i.e. speed) is an important requirement for these projects (as evidenced by the fact that they all have targeted parallel supercomputers), other non-functional requirements are considered just as important in the long run. Performance is important only to the extent that the software can be used by its customers to meet their deliverables and milestones. The longevity of these projects, along with the relative frequency of new computing platforms, necessitate that *portability* and *maintainability* be considered along with *performance* as important, and often competing, goals. The ability to easily port a project to new machines increases the likelihood of its long-term success. Conversely, if a team spends a large amount of effort tuning the performance for one specific platform, that effort is wasted when a new platform becomes available prior to the release of the software. Members of the CONDOR team highlighted this issue in the following comment:

> *People want the environment [provided by CONDOR] on their laptops, their workstations and then log onto an HPC center on the other coast, put it there, run it, and then take the results back down to their workstations.*

Overall the most highly ranked project goals were:

1. Correctness
2. Performance
3. Portability
4. Maintainability

Of these goals, *portability* was the only one that was ranked as having "high" importance by all of the teams studied.

The implication of this lesson is that the software engineering methods needed in scientific and engineering software may be different from those needed for more traditional IT software. Methods must be chosen and tailored so they are properly aligned with the software development goals.

### 5.7 Agile methodologies are better accepted by scientific and engineering code developers than more traditional methodologies.

Scientific and engineering projects appear to lend themselves to the use of agile software development approaches. In this case, "agile" refers to the philosophical approach of agile methods rather than to any particular method, such as eXtreme Programming. Because many of these projects are doing new science and the requirements are not (and cannot be) known in detail in advance, operating in a rigid plan-driven style (as some sponsors require) is not feasible or productive. These teams need the flexibility to experiment with different methods quickly to find ones that work. While some form of planning is essential to success, rigid software management is avoided.

Most of the teams studied have been successfully operating with an agile philosophy (although they did not always realize it) for decades -- long before the term "agile" entered the software engineering vocabulary. These teams have tended to favor individual team members and good practices over more rigid processes and tools.

The implication of this lesson is that existing software engineering development methodologies and philosophies need to be tailored for scientific and engineering software development. Rigid, process-heavy approaches tend not be used, both for technical reasons (i.e. unstable or unknown *a priori* requirements) and cultural reasons (i.e. the developers, who tend to be scientists rather than engineers, tend to view "process" unfavorably).

### 5.8 Multi-disciplinary teams are important to the success of these projects.

The staffing profiles of these projects show the multi-disciplinary nature of the problems being solved. In total, computer scientists make up less than 20% of the team members (although on different teams that percentage varies from 0% - 33%), with the rest being domain scientist or engineers. The main reason for this staffing profile is presence of two types of complexity: domain complexity and software complexity.

Domain complexity is evident in the fact that much of this software is written to simulate highly complex physical or engineering behavior. In fact, many of the applications require a PhD in physics or a branch of engineering just to understand the problem. The teams have found it easier, and more practical, for the domain scientists and engineers to learn how to write software than for software engineers to learn all of the relevant science or engineering concepts. Conversely, to achieve performance and flexibility in such complex applications, the teams also are in need of software engineering expertise. A member of the HAWK team put it this way:

> *In these types of high performance, scalable computing [applications],* in addition to the physics and mathematics, *computer science plays a very major role. Especially when looking at optimization, memory management and making [the code] perform better…You need a multi-disciplinary team. It [C++] is not a trivial language to deal with…You need an equal mixture of subject theory, the actual physics, and technology expertise.*

### 5.9 Success or failure of the project depends on keeping customers satisfied (in addition to sponsors).

This lesson is not unique to computational science and engineering, but it is important to understand it within the context of the scientific and engineering software community. These projects appear to have a different business model than for more traditional commercial IT software. Funding often comes from a governmental agency while the customers may or may not be part of that same agency. Therefore, the success of the project depends on satisfying both constituencies. In the long run, keeping the customers happy by responding to their needs is an important factor in the success of any project. But, even if the development team meets all the requirements and milestones set by the sponsor, it may not succeed without a supportive user community. For example, because of the lack of users, HAWK has been suspended despite its success technically. Balancing the needs of all stakeholders can be a challenge.

## 6. Summary

This paper discussed five case studies of scientific and engineering software development projects sponsored by the US Department of Defense, Department of Energy, and National Science Foundation. Each project came from a different scientific or engineering domain with different overall goals. Based on the cross-analysis of these projects, we presented a series of lessons learned in Section 5.

Overall, the work described in this paper led to the following conclusions. First, we provided some insight into the difficulty of verification and validation.

Second, we highlighted why agile development philosophies closely fit these teams. Third, we came to an understanding of why these projects do not adopt higher-level languages, like Matlab, when doing so would appear to provide advantages. Fourth, we highlighted that while performance is important in this domain (hence the use of parallel supercomputers) it is often not the most important goal. Fifth, we explained that there is an expertise gap because of the complexity of the code and difficulty of the underlying domain. Sixth, we showed why these software development teams avoid IDEs. Seventh, we explained why teams favor tools developed in-house (or open-source) rather than commercial third-party tools. Each lesson was supported with specific observations.

The goal of this paper is to provide information that can be useful for the software engineering community as well as the computational science and engineering community. For the software engineering community, this paper highlighted some of the reasons why the development process for this type of software is different from the development process for commercial IT software and why some of the traditional software engineering approaches have not been adopted. Some of these differences are inherent to the domain, while others could be addressed through education (e.g. learning a new IDE). In addition, we have highlighted some instances where existing software engineering research is inadequate for this domain. For the computational science and engineering community, this paper provided some insights that could guide the improvement of the software engineering process.

## 7. Acknowledgements

We would like to thank the members of the project teams. We would also like to thank case study team members Andy Mark, Christine Halverson, Dolores Shaffer. This research was supported in part by the United States Department of Energy (DOE) contract DE-FG02-04ER25597 to USC-ISI and DOE contract DE-FG02-04ER25633 and Air Force grant FA8750-05-1-0100 to the University of Maryland.

## 8. References

[1] Card, D.N., Church, V.E., Agresti, W.W., "An Empirical Study of Software Design Practices." *IEEE Transactions on Software Engineering*, 1986. **12**(2): 264-271.

[2] Hochstein, L., Carver, J., Shull, F., Asgari, S., Basili, V., Hollingsworth, J.K., and Zelkowitz, M. "HPC Programmer Productivity: A Case Study of Novice HPC Programmers". In *Proceedings of SuperComputing*. 2005. p. 35

[3] Johnson, P. "Workshop on software engineering for high performance computing system (HPCS) applications". In *Proceedings of Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*. 2004. p. 772-772

[4] Johnson, P.M. "Second international workshop on software engineering for high performance computing system applications". In *Proceedings of Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*. 2005. p. 683-683

[5] Kendall, R.P., Carver, J., Mark, A., Post, D., Squires, S., and Shaffer, D. *Case Study of the Hawk Code Project*. Technical Report, LA-UR-05-9011. Los Alamos National Laboratories: 2005.

[6] Kendall, R.P., Mark, A., Post, D., Squires, S., and Halverson, C. *Case Study of the Condor Code Project*. Technical Report, LA-UR-05-9291. Los Alamos National Laboratories: 2005.

[7] Kendall, R.P., Post, D., Squires, S., and Carver, J. *Case Study of the Eagle Code Project*. Technical Report, LA-UR-06-1092. Los Alamos National Laboratories: 2006.

[8] Müller, M.M. and Tichy, W.F. "Case study: extreme programming in a university environment". In *Proceedings of 23rd International Conference on Software Engineering*. May 12-19, 2001. p. 537-544

[9] Perry, D.E., Sim, S.E., and Easterbrook, S.M. "Case studies for software engineers". In *Proceedings of Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*. 2004. p. 736-738

[10] Post, D.E. and Kendall, R.P., "Software Project Management and Quality Engineering Practices for Complex, Coupled Multiphysics, Massively Parallel Computational Simulations: Lessons Learned From ASCI." *International Journal of High Performance Computing Applications*, 2004. **18**(4): 399-416.

[11] Post, D.E., Kendall, R.P., and Whitney, E. "Case study of the Falcon Project". In *Proceedings of Second International Workshop on Software Engineering for High Performance Computing Systems Applications (Held at ICSE 2005)*. St. Louis, USA. 2005. p. 22-26

[12] Seaman, C.B. and Basili, V.R. "An Empirical Study of Communication in Code Inspections". In *Proceedings of 19th International Conference on Software Engineering*. Boston, MA. May 17-23, 1997. p. 96-106

[13] Shull, F., Carver, J., Hochstein, L., and Basili, V.R. "Empirical Study Design in the Area of High Performance Computing (HPC)". In *Proceedings of International Symposium on Empirical Software Engineering*. Noosa Heads, Australia. 2005. p. 305-314

[14] Van De Vanter, M.L., Post, D., and Zosel, M.E. "HPC Needs a Tool Strategy". In *Proceedings of Second International Workshop on Software Engineering for High Performance Computing System Applications (held at ICSE 2005)*. St. Louis. 2005. p. 15

[15] Yin, R.K., *Case Study Research : Design and Methods (Applied Social Research Methods)*. 3rd ed. 2002: SAGE Publications.