ABSTRACT

Title of Dissertation: DEVELOPMENT OF AN EMPIRICAL APPROACH TO BUILDING DOMAIN-SPECIFIC KNOWLEDGE APPLIED TO HIGH-END COMPUTING

Lorin Hochstein, Doctor of Philosophy, 2006

Directed By: Professor Victor R. Basili
Department of Computer Science

This dissertation presents an empirical approach for building and storing knowledge about software engineering through human-subject research. It is based on running empirical studies in stages, where previously held hypotheses are supported or refuted in different contexts, and new hypotheses are generated. The approach is both mixed-methods based and opportunistic, and focuses on identifying a diverse set of potential sources for running studies. The output produced is an experience base which contains a set of these hypotheses, the empirical evidence which generated them, and the implications for practitioners and researchers. This experience base is contained in a software system which can be navigated by stakeholders to trace the "chain of evidence" of hypotheses as they evolve over time and across studies.

This approach has been applied to the domain of high-end computing, to build knowledge related to programmer productivity. The methods include controlled

experiments and quasi-experiments, case studies, observational studies, interviews, surveys, and focus groups. The results of these studies have been stored in a proof-of-concept system that implements the experience base.

DEVELOPMENT OF AN EMPIRICAL APPROACH TO BUILDING DOMAIN-SPECIFIC KNOWLEDGE APPLIED TO HIGH-END COMPUTING

By

Lorin Hochstein

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2006

Acknowledgements

The completion of this dissertation would not have been possible without the help and support of many people. First, I would like to thank my parents who instilled within me the importance of education, and my brother, for always being there.

Empirical studies are always a team effort, and the many studies described in this work could not have been conducted without the combined effort of researchers at the University of Maryland and at the Fraunhofer Center: Vic Basili, Marv Zelkowitz, Forrest Shull, Jeff Hollingsworth, Sima Asgari, Taiga Nakamura, and Patricia Costa. I am particularly indebted to my advisor, Vic Basili, for his boundless optimism and his tireless support of this work, even when it seemed (to me, at least) that so little progress was being made.

I would also like to thank the professors who were willing to allow their classrooms to be used as software engineering research laboratories: Henri Casanova, Jacqueline Chamé, Alan Edelman, John Gilbert, Mary Hall, Jeff Hollingsworth, Glenn Luecke, Ed Luke, Aiichrio Nakano, Allan Snavely, Alan Sussman, and Uzi Vishkin.

I am grateful to the members of my dissertation committee: Vic Basili, Marv Zelkowitz, Jeff Hollingsworth, Alan Sussman, and Bill Dorland. Their time and feedback have improved the quality of this work.

Thanks go to the undergraduate students from the University of Mannheim who worked with me: Thiago Craverio, Patrick Borek, Nico Zazworka and Martin Voelp.

I would like to thank Jeff Carver for teaching me so much about how to design and run empirical studies. I learned by watching him.

I would like to thank all of the participants of my empirical studies. This work would not have been possible without so many people who were willing to participate in this work.

I would like to thank Mikael Lindvall who spent much time working with me at the beginning of my academic career. Alas, I failed to heed his advice to do something "quick and simple", and followed his example instead.

Finally, I would like to thank my wife, Stacy, who supported me throughout the entire process and always helped me find the strength to continue.

Table of Contents

List of Tables

List of Figures

Chapter 1 Introduction

1.1 Overview

Traditionally, research in software engineering has focused on what Glass calls *advocacy research*: developing and promoting the use of new technologies [Glass94]. By contrast, there has been significantly less research on understanding the impact of technologies [Fenton93] or the current state-of-the-practice [Glass03]. Understanding how technologies are actually used by practitioners, and, more generally, how they carry out the software development process, is a challenge. Software development is inherently a human-centric process, and human-subject research involves many complexities that are not present in other types of research [Basili06]. While other domains such as medicine, psychology, and sociology, have developed methods for managing these complexities, empirical software engineering research is still a young field, and continues to develop appropriate tools and methods.

Another challenge in software engineering research is the enormous degree of variation across software projects. For example, the software which controls a vending machine is very different from the software which controls an e-commerce website, which in turn is different from the software which controls a nuclear power plant. Given the differences in requirements, we expect the software development processes to look very different in terms of tools, methodologies, and programmer skill sets. Yet, a survey of the literature revealed that only 2.7% of software engineering papers dealt with problem-domain-specific-issues [Glass02].

One such domain is high-end computing (HEC), which is also referred to as high-performance computing (HPC) or supercomputing. This domain is generally synonymous with computational science, where the goal is to study physical phenomena through computer simulation. For many problems of interest, desktop computers are not capable of running such simulations, so scientists must write software to run on more powerful, parallel machines.

The process of writing, executing, and evaluating software, on esoteric computer hardware, intended to produce novel scientific results, presents many challenges to the computational scientist. Consider the following quotes from reports commissioned by the U.S. government:

- "*Today, it is altogether too difficult to develop computational science software and applications*" [PITAC05]
- *"There is widespread agreement that trends in both hardware architecture and programming environments and languages have made life more difficult for scientific programmers"* [Graham04]
- *"Applications areas have productivity problems because the time to program new supercomputers is increasing."* [HECRTF04]
- "*When asked what could be accomplished if the 'ease-of-use' barrier were addressed with systems that are 10 times easier to program, respondents overwhelmingly indicated that they could develop more powerful applications and fundamentally rewrite their current codes*." [Joseph04].

The consensus in the community is that it is simply very difficult to write applications ("codes" in the HEC jargon) on HEC systems, and this presents a significant barrier to the productivity of scientists and engineers. The recurring message is that today's technologies are inadequate to HEC users, and that the problem is getting worse over time.

Many solutions have been proposed by researchers and technologists to address this problem, such as improving:

- programmers (education/training)
- software development methodologies (better processes)
- algorithms (research into new algorithms)
- compilers (research into parallel compilers)
- languages (new parallel programming languages)
- hardware (better architectures)
- tools (new libraries, debuggers, profilers, IDEs, etc.)

As an example, the ongoing DARPA High Productivity Computing Systems[1] (HPCS) project is focusing largely on new hardware and new languages as the solution to the productivity problem.

However, despite wide recognition of the software engineering problems, there has been little previous empirical software engineering research into what aspects are most difficult and would be most amenable to interventions such as those mentioned above. As noted by Pancake and Cook, "*Applicable information on [HEC] programming habits is remarkably absent from the computing literature. There is a substantial body of work derived from empirical studies of programmers, but the*

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

subjects, materials, and tasks differ dramatically from the parallel user community in key ways" [Pancake94].

1.2 The problem

A major assumption of this dissertation is that we must *empirically investigate* proposed technologies if we wish to study their impact on programmer productivity, and this will necessarily involve human-subject research. For example, the impact of a tool on a given task can only be understood if we know how users typically go about the task without the tool [Pancake94]. Given that so many variables can affect the software development process, any such investigation into technology must also take into account the impact of context variables, such as programmer background and project size.

Furthermore, given the lack of previous research in the domain of high-end computing, we believe it would be shortsighted to focus on evaluating a single type of technology. In fact, "productivity" in high-end computing does not have a single accepted definition in the community [Kepner04]. There is a clear need for fundamental research into the software development process in this domain to better understand where programmers have problems and how well proposed technologies would help solve these problems.

There are many challenges to undertaking such research, such as:

- lack of an existing body of software engineering research in the domain
- difficulty in obtaining access to subjects and projects from the populations of interest

- relatively little domain knowledge on the part of the empirical researcher

Based on this, we can state the central research problem addressed in this dissertation: **How do we build knowledge empirically in a software domain where little software engineering research has previously been done?**

1.3 Proposed solution

This dissertation presents an approach for conducting this research using empirical, human-based methods. The approach is an *iterative* methodology. It allows researchers to build knowledge incrementally, in a layered fashion: successive studies build on top of previous ones as existing hypotheses becoming more clearly defined and supported, and new hypotheses are generated. It is also an *opportunistic* methodology, drawing on all of the available resources to try and cover a large range of study designs and context variables. We use as an example of this process our current involvement in the DARPA HPCS program with its goal of understanding and improving programmer productivity in the HEC domain.

1.4 Organization of the dissertation

The dissertation is organized as follows: Chapter 2 provides background information and describes related work. Chapter 3 lays out the proposed methodology that addresses the research problem. Chapter 4 provides an overview of the studies conducted as a result of applying the methodology. Chapter 5 describes how the methodological infrastructure evolved across the studies in accordance with the methodology. Chapter 6 provides examples of empirical knowledge that evolved as the studies progressed. Chapter 7 introduces a framework for organizing the results into an experience base. Chapter 8 concludes the dissertation with a summary of contributions, lessons learned, and future avenues of research.

Chapter 2 Background and related work

2.1 Survey of programming models and technologies

Since compilers today are not sophisticated enough to take a sequential program and run it efficiently on a high-end (i.e. parallel) machine, special-purpose technologies must be used for programming such machines There are several technologies available today for programming parallel machines. Parallel programming technologies can be roughly characterized by the *parallel programming model* that they support. It is through the programming model that the programmer specifies how the different processes in a parallel program coordinate to complete a task (In general, when a program runs on *N* processors of a parallel machine, *N* processes will be launched, one running on each processor). The most common models currently in use are:

- Message passing
- Threaded
- Global arrays
- Data-parallel

Other parallel programming models exist (e.g. PRAM [JáJá92], dataflow [Gurd85]), but are not commonly supported on current parallel architectures.

2.1.1 Message-passing

In the message-passing model, each process executes within a separate address space, and processes communicate by exchanging messages. Communication is “two-sided”: to communicate, one process must issue a “send” request and another must issue a matching “receive” request.

Communication can be either blocking or non-blocking. If communication is blocking, then a “send” request will not complete until some particular event has occurred (e.g. the data has been copied to a buffer, the sender has received an acknowledgement that the data has been received). If communication is non-blocking, then the sending process can make a “send” request and then continue to execution even though the request might not have completed. Non-blocking calls are used to improve performance of a parallel program by overlapping computation and communication. However, non-blocking communication can increase the complexity of the code. The programmer has to consider which calculations can be done while messages are in transit, and re-organize the code accordingly.

Message-passing is generally suited for coarse-grained parallelism, because of the communication overhead that the model incurs. The process of breaking up a problem into sub-problems that can be run in parallel (known as “domain decomposition”) can be particularly challenging in the message-passing model.

Examples of message-passing technologies include MPI [Dongarra96] and PVM [Sunderam94]. MPI is currently the most popular parallel programming technology. MPI advocates claim advantages such as [Gropp94]:

- Universality (works on all different parallel machines)
- Expressivity (can express different kinds of parallel algorithms)
- Ease of debugging (controls memory references explicitly so unexpected

overwriting of memory is less common)

- Performance

While MPI is the most popular parallel programming technology in terms of number of users, it is not well-liked. MPI is considered difficult to program compared to serial programming. In particular, MPI forces programmers to work at a very low-level of abstraction and deal with many of the communication details. Yet, this is also considered one of the strengths of MPI, as the programmer has a great deal of control over the communication between processes.

2.1.2 Threaded

In the threading model, all of the processes execute in a shared address space, and are referred to as *threads*. Since all processes can access all elements of memory, no explicit communication is needed to pass data from one process to another. Unfortunately, it is possible to write multithreaded programs which are non-deterministic, where the output of the program depends upon the order in which the threads are scheduled[2]. For example, if two threads update a given variable, the value of the variable may depend upon the timing of the threads, which can change from one run of the program to the next. These phenomena are referred to as *races* (where the behavior of the program depends upon which thread "wins the race" through the code) and are always undesirable. Therefore, synchronization mechanisms are necessary to ensure that programs run deterministically. All languages and libraries that support the threaded model provide synchronization constructs to prevent race

[2] Non-determinism can occur in message-passing programs as well.

conditions. It is the responsibility of the programmer to use these constructs in the appropriate places to generate a deterministic program. If too few constructs are used, the program may function correctly in some runs but not in others. If too many synchronization constructs are used, performance will suffer because of threads being blocked in synchronization calls rather than doing useful work.

Examples of threaded technologies are OpenMP [Dagum98] and pthreads [Narlikar98], with OpenMP being the more popular threading technology for HEC programming [NSF05]. OpenMP advocates claim advantages such as [Chandra01]:

- Ability to parallelize small parts of an application at a time
- Simple parallel algorithms are easy and fast to implement
- Small impact on code size and readability compared to serial

OpenMP is believed to require significantly less effort than MPI for implementing parallel programs. However, OpenMP is also believed to only be suitable for symmetric multi-processor (SMP) machines, which are currently limited to hundreds of processors. By contrast, the largest known machine to date, BlueGene/L, contains over one hundred thousand processors[3].

2.1.3 Global arrays

In the global array model, each process resides in a separate access space, as in the message passing model. However, a process can access a restricted set of memory elements in the address space of other processes as well. This model supports "one-

[3] http://www.top500.org

sided" communication, where one process can send data to another without a matching receive call.

Examples of global-array technologies are Co-Array Fortran (CAF) [Numrich98], Unified Parallel C (UPC) [Carlson99], Titanium [Yelick98], and the GA Toolkit library[Nieplocha96]).

Global array languages are relatively new. There is considerable interest in the HEC community about whether these languages can reduce the effort compared to MPI and achieve good performance on large-scale systems.

2.1.4 Data-parallel

In data-parallel technologies, the programmer does not specify the parallelism explicitly. Rather, the parallelism is extracted by the compiler or run-time system, and the programmer provides hints as to how the data should be distributed. Examples of data-parallel languages include High Performance Fortran [Bokus94], ZPL [Chamberlain98], MATLAB*P [Choy03], and HPC++ [Johnson97].

The data-parallel model is considerably less popular than the message-passing or threaded models. Of particular note is High Performance Fortran, which failed to achieve widespread adoption in spite of a large standardization effort by the HEC community [PITAC05]. Two of the problems associated with High Performance Fortran are insufficient support for a broad range of problems (e.g. irregular problems, task parallelism), and lack of *performance transparency*, which makes it difficult to determine how to improve the performance of a particular code. Other data-parallel technologies may be prone to similar problems.

2.1.5 Hybrid approaches

The models described above are not necessarily mutually exclusive. For example, it is possible to write hybrid programs that combine MPI and OpenMP, or MPI and Co-Array Fortran. Hybrid approaches are applied to achieve better performance than a strict MPI implementation by avoiding the overhead of message-passing when communication among processors within an SMP node.

2.2 Related work

2.2.1 Decision support for HEC practitioners

One of our goals is to help HEC stakeholders make decisions about which technologies they should use. Some researchers have offered guidelines to help practitioners decide when and how to parallelize their programs. Pancake lists three preconditions that should be considered before deciding to try and use parallelism [Pancake96]:

1. How frequently will the application be used before changes are needed?
2. How much time is currently needed to execute the application?
3. To what extent are you satisfied with the current resolution or complexity of your results?

A recent book by Dongarra et al. has an overview of the different parallel programming technologies and provides the following guidelines for choosing whether or not to use a particular technology [Dongarra03]:

Use	**If**
Parallelizing compilers	• Goal is to extract moderate O(4-10) parallelism from existing code • Portability is not a major concern
OpenMP	• Goal is to extract moderate O(10) parallelism from existing code • Good quality implementation exists for target platform • Portability to distributed-memory platforms is not a major concern
MPI	• Scalability is important • Application must run on some message-passing platforms • Portability is important • A substantive coding effort is acceptable to achieve other goals
PVM	• All MPI conditions plus fault tolerance are needed
HPF	• Application is regular and data parallel • A simple coding style in Fortran 90 is desirable • Explicit data distribution is essential to performance • A high degree of control over parallelism is not critical
Co-Array Fortran	• An implementation is available
Threads	• Scalability is not important • Program involves fine-grained operations on shared data • Program has significant load imbalances • OpenMP is not available or suitable
CORBA,RMI	• Program has task-parallel formulation • Interested in running in network-based system • Performance is not critical
High-level libraries	• They address your specific problem • The library is available on the target platform

(Note: CORBA and RMI are typically used only in distributed systems, which are not discussed in this work).

2.2.2 Empirical studies in HEC

There have been few empirical studies into programmer productivity in high-end computing. A notable exception is the survey work of Pancake [Pancake94, Pancake96, Pancake97, Pancake00]. There have also been some empirical studies to quantify the effect of parallel programming technologies on effort. Szafron & Schaffer performed a controlled experiment to compare the Enterprise parallel programming system to a message-passing library in terms of their usability [Szafron96]. Rodman and Brorsson performed an experiment to analyze the effort required in improving the performance of a shared memory program by adding message-passing constructs to performance-critical sections [Rodman99]. Additionally, there has been some evaluation of parallel programming technologies through source code analysis [Cantonnet05, Chamberlain00].

2.2.3 Case studies in HEC

Some case studies of HEC projects have been documented in the literature. Berthou and Fayolle performed a case study where they parallelized a program using three separate programming technologies: MPI, OpenMP, and HPF [Berthou01]. Morton discussed the effort-performance tradeoffs in porting a PVM program from an IBM RS/6000 cluster to a Cray T3D [Morton95]. More recently, Post has done some post-mortem case study research on HEC projects, focusing on project-level issues such as requirements, scheduling and management [Post03].

2.2.4 Productivity models in HEC

The HEC community is shifting from a focus on machine performance to a more general notion of productivity that incorporates hardware, software, administrative and maintenance costs [Kepner04]. However, as noted by Snir and Bader, the traditional economic notion of productivity (output divided by cost) cannot be applied in this domain because the output of an HEC system is difficult to measure [Snir04]. They propose that utility theory be applied to measure the output of an HEC system, and define productivity as utility divided by cost, where utility is a function that represents the preference of a rational agent (e.g. lab director), and is generally a decreasing function of time-to-solution. Their model of productivity is on a per-system and per-problem basis.

Sterling proposes a similar notion of productivity as output over cost for a machine, although his measure is over the lifetime of the machine rather than on a per-problem basis [Sterling04]. He describes output as a set of results produced by the system, weighted by the subjective value of these results. Since it is not obvious how these weights could be determined, he proposes measuring the "useful work" of the machine as a proxy: the number of basic calculation operations, excluding tasks such as managing parallel resources, duplicated work, overhead, etc. Kuck has also suggested counting results produced per year, although he admits that this is difficult to define precisely [Kuck04].

Faulk et al. propose measuring output as a vector, where the different elements relate to different quality attributes of the software being developed (e.g.

performance, portability, maintainability). Each element in the vector is a product of two values: how much the stakeholder values the quality attribute, and how well the software realizes that attribute. The elements of the vector can be summed to provide a total measure of value. [Faulk04]

Kennedy et al. consider productivity on a per-language basis rather than a per-system basis [Kennedy04]. They characterize a language by two variables: efficiency and power. Both of these variables are defined relative to a reference language on a reference system, and can be computed for a particular application. Efficiency is defined as the ratio of development time in the reference language to development time in the language being evaluated. Power is defined as the ratio of execution time in the reference language to execution time in the language being evaluated. They define productivity as the ratio of time-to-solution in the reference language to time-to-solution in the evaluation language, and recast this as a function of efficiency, power, and the number of times the program will be executed. Zelkowitz et al. propose a similar productivity model which measures productivity of a language as the ratio of speedup to effort, both of which are defined as relative to a reference serial implementation [Zelkowitz05].

2.2.5 Productivity models in software engineering

The software engineering community has built up a significant body of work on analyzing the effect of different variables on programming effort, although none of this work has focused specifically on parallel programming. While there has been considerable research in the software engineering community on productivity, the notion of productivity in software engineering is related entirely to effort, and is usually measured in units such as “lines of code per hour” or “function points per hour” [IEEE-1045]. Some examples of parametric models related to productivity are Boehm’s COCOMO model for predicting the effort required to complete a project based on an estimate of code size, which is adjusted based on several other variables [Boehm81], Bailey-Basili’s meta-model [Bailey81] and the Walston-Felix model. [Walston77].

2.2.6 Combining evidence from multiple sources

Many researchers have examined the problem of combining evidence generated from multiple studies. Of particular note is the Evidence-Based Medicine movement where medical practitioners make decisions based upon a rigorous assessment of the related studies [Strauss05]. Medical researchers developed the *systematic review* process for surveying the literature to locate relevant evidence when making diagnostic decisions [Cook97].

Meta-analysis is one approach for accumulating knowledge by combing the results of multiple studies [Schmidt92, Miller00]. Meta-analysis is typically performed by identifying a set of published studies that are concerned with measuring a particular effect and combining the results from these studies. The goal of meta-analysis is to form a more accurate estimate of the size of the effect under investigation. Several researchers have discussed the use of meta-analysis in software engineering research [Brooks97, Pickard98, Miller99, Miller00].

Such meta-analytic approaches are traditionally focused on combining results from controlled experiments, and not across studies of different types (e.g. combined controlled experiments, case studies, and surveys). However, some researchers have attempted to combine results from multiple studies for other purposes, such as identifying the role of context variables [Shull05].

The problem addressed in this dissertation is not that of identifying relevant studies that have been performed previously and synthesizing the results. Rather, the problem here is that there is an absence of an existing body of work to build upon, and that new studies must be designed and conducted. Given that the results being generated are completely new, the researcher has the opportunity to store them in a format other than simply the archival literature, such as an *experience base* or *experience factory,* proposed by Basili [Basili89, Basili94, Basili01]. Basili proposed that organizations which collect empirical data on projects and processes should store and manager their data in an appropriate repository which supports incremental growth and decision-making.

2.2.7 Empirical methods

While empirical software engineering is a relatively young research area, there exists a substantial body of knowledge on empirical methods, heavily borrowed from domains such as psychology, anthropology, and medicine, especially with regards to the human-based aspects of empirical software engineering [Basili06]. A full review of empirical methods is beyond the scope of this work. For the interested reader, Robson provides a comprehensive review of the research methods available to social scientists [Robson02]. We provide a brief overview of commonly used designs and data collection methods.

Designs

Many study designs have been developed by researchers. Some of the more common designs used in software engineering are:

- Controlled experiments and quasi-experiments
- Correlational studies
- Case studies
- Ethnographic studies

In a controlled experiment or quasi-experiment, the researcher seeks to determine if there is a relationship between two variables, an independent variable and a dependent variable. The subjects are divided up into two or more groups[4], and the value of the independent variable is systematically varied across the groups. (In software engineering the independent variable is often a technology, such as an inspection technique or a debugger, and the dependent variable is some outcome variable of interest, such as programmer effort or number of defects found). A statistical test is applied to determine whether the technology had a measurable effect. Campbell and Stanley describe different experimental designs [Campbell63] and Kitchenham provides software-engineering specific guidelines for conducting experiments [Kitchenham02].

In a correlational study, the researcher does not control the independent variable. Instead, the researcher selects a population where the independent variable varies

[4] Assignment is random in an experiment, non-random in a quasi-experiment.

naturally, and uses correlational analysis to measure the relationship between the independent and dependent variables.

In a case study, one or more “cases” are studied in detail. According to Yin, a case study “investigates a contemporary phenomenon within its real-life context when the boundaries between the phenomenon and context are not clearly evident” [Yin94]. The case study method is primarily qualitative. Typically, a case study involves collecting and synthesizing data from multiple sources (e.g. interviews, documents, observations). Perry et al. have advocated the use of the case study method in software engineering research [Perry04]. Seaman provides an overview of qualitative approaches which can be applied in case studies [Seaman95].

In an ethnographic study, the object of study is the culture and social structure of a social group. Traditionally, such studies require that people are studied for long periods of time in their natural environment [Robson02]. Like the case study, the ethnographic study relies largely on qualitative methods.

Data collection methods

For each type of study, data must be collected that relates to the subjects. There are various methods for collecting such data. Some common ones include:

- Surveys
- Interviews
- Diaries
- Observation
- Unobtrusive measures

It is assumed that the reader is familiar with the general concept of surveys and interviews. *Surveys* can be delivered in several ways (e.g. verbally, paper form, web-based form).

Interviews are often classified by the degree of structure in the interview guide used by the interviewer [Robson02]. An interview may be *structured* (no deviation from the guide), *semi-structured* (the interviewer may depart from the guide in response to the subject) or *unstructured* (no guide at all).

When *diaries* are employed, the subjects being studied record information into a diary or log, which is later collected and analyzed by the researcher. The level of structure in a diary may vary from free-form to a structured log, where the subject is essentially filling out a form.

In *observation*, the researcher becomes the measurement tool, observing the subjects directly. The researcher's role may vary, from being a *fly-on-the-wall* (no interaction with subjects) to being a *participant observer* (researcher as co-worker) [Lethbridge05]

Unobtrusive measures do not require any additional actions on the part of the subject to collect the data. The researcher may analyze data that is normally generated by the subject (e.g. documents), or the researcher may introduce some additional data collection mechanisms that are transparent to the subject.

Chapter 3 Methodology

This chapter motivates and describes a methodology for conducting empirical software engineering research in software domains not previously studied from a software engineering point of view. This methodology was instantiated to study programmer productivity in high-end computing. The structure of this chapter is modeled largely on [Vegas02].

As discussed in Chapter 1, the following aspects are true about the domain of high-end computing.

- Developing software for HEC systems is very difficult.
- Existing programming models are perceived as being a great obstacle to programmer productivity.
- The notion of "productivity" is not well-defined.

According to Chapter 2, the state of the art with regard to productivity research in HEC can be summarized as follows:

- General guidelines exist for selecting programming models, which are based on practitioner experience.
- Very few studies have been done to compare programming models directly.
- Few, if any studies have been performed to explore other issues that relate to programmer productivity in this domain.
- In general, very little empirical software engineering research has been done in this particular domain.

Given the widespread view that developing HEC software is very difficult, and that so little software engineering research has been done in this domain, there is a clear need for further research in this area. This research is all the more challenging given the large number of context variables, and the finite resources of the researcher. While these challenges are not insurmountable, the task of developing a research methodology to address this problem *is itself a research problem of empirical software engineering.*

The approach chosen in this dissertation is to build knowledge incrementally, gaining maturity in experimental methods and gradually branching out to examine different factors in more detail and different types of projects. The general problem of empirical knowledge building in a software domain can be defined as the composition of three problems:

1. Identifying particular phenomena in the domain to be investigated
2. Designing and conducting a set of studies
3. Presenting the results to stakeholders in a useful format

For the first problem, identifying the particular phenomenon of interest is highly dependent upon the domain knowledge of the researcher. One of the major challenges here is simply *being able to ask the right questions*.

For the second problem, the ability of the researcher to conduct studies is highly dependent upon the resources available, especially in terms of human subjects.

For the third problem, it is essential that the knowledge generated by the studies be disseminated to the stakeholders (e.g. developers, technologists, educators) in a form they can make use of. The traditional method of publishing results in the archival literature is generally not an effective way of reaching practitioners

[Dyba05].

Our goal is to **develop an empirical methodology for building an experience base that captures useful knowledge about a particular software domain**. It follows that the researcher must take three approaches to solve the methodological problem: build sufficient domain knowledge to identify the phenomena that should be studied in more detail, locate the appropriate resources so that studies can be run, and create an organizational framework for capturing the results. These are the three elements that this research aims to address.

3.1 Description of the problem

As concluded from the overview of related work, while there have been a few individual studies in the domain of high-end computing, there has not previously been a systematic methodology developed for conducting the research in this domain (or any other specific domain of software engineering).

Although there is no existing methodology at present, the characteristics of the methodology are known:

- Researchers who are not necessarily experts in the domain, but are familiar with empirical methods, should be able to implement the methodology.
- It should support the generation of quantitative results, which are important for certain kinds of decision-making.
- It must analyze realistic software projects in the domain.
- The resources required by the methodology must be accessible to researchers.
- The methodology must support an incremental building of knowledge.

- The methodology should support conducting of studies by domain researchers who are not experts in empirical software engineering research.
- Stakeholders must be able to identify the origins of the knowledge for gauging their confidence in the knowledge and understanding where it originated.

In essence, the research problem is defined by the requirements above. It is reasonable to expect that the actions to be taken by a researcher who wishes to build knowledge in a domain will follow the basic steps of conducting empirical research, which are:

1. Identify the phenomena of interest to be studied
2. Generate testable hypotheses
3. Design and conduct appropriate studies to test hypotheses
4. Add results to the collective body of knowledge
5. Repeat from step 1

However, this process is not as straightforward as it appears:

- When conducting research in a new domain, the researcher may not know what phenomena should be investigated. For example, when exploring productivity bottlenecks, it would be short-sighted to focus only on issues related to programming languages.
- Generating testable hypotheses in software engineering can be very difficult because of the very large number of context variables that can have influence over the phenomena of interest (e.g. effort, performance, defects).
- There are a number of constraints on the types of studies that can be

conducted by empirical software engineering researchers, due to limits on available resources. In particular, it can be difficult to obtain access to subjects with experience in HEC programming, as well as obtaining access to HEC projects. In addition, the researcher may not have access to the appropriate tools for carrying out the research.

- It can be difficult to integrate the results from multiple studies when the studies use different methods and have been run in different environments. Simply publishing the result of a particular study without placing it in context of the accumulated body of knowledge makes it difficult for stakeholders to apply the knowledge to make decisions.

In contrast to this idealized process for conducting research, there is the typical process employed by researchers conducting studies such as those mentioned in Chapter 2. This process is characterized as follows:

- A researcher encounters an opportunity to conduct a study on some aspect of HEC of interest.
- The researcher conducts the study.
- The results are published in the archival literature, but the knowledge is not explicitly placed within the context of the larger body of knowledge.
- No follow-up work is explicitly done by the individual researcher or other researchers.

We envision stakeholders making use of knowledge built by researchers, as follows. The role of the stakeholders is to apply the existing body of knowledge to improve programmer productivity in practice. The specific action taken will depend upon the stakeholder. For example, programmers may apply the knowledge to select tools for a project, technologists may apply the knowledge to develop new tools, and professors may apply the knowledge to modify the material presented in courses. The role of the researchers is to build knowledge by running successive studies and accumulating evidence.

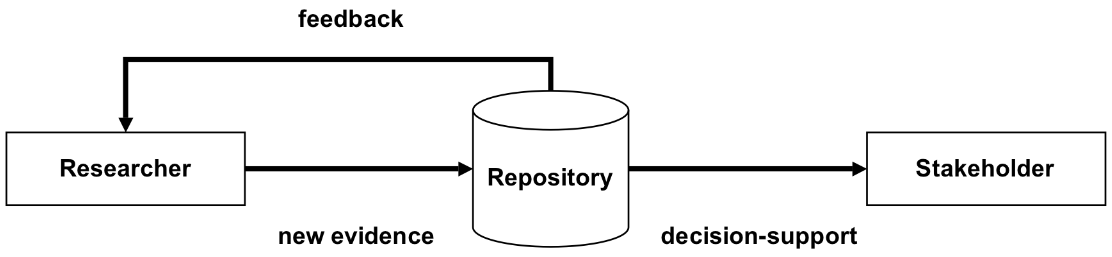


Figure 3.1 Flow of evidence

From all this, it follows that the problem to be solved involves the development of a methodology that will be able to be used, on the one hand, to conduct domain-specific research in a new software domain and, on the other, to produce a body evidence that can be navigated by stakeholders to be used for decision-making.

3.2 Problem-solving approach

The solution proposed in this research to the problem of conducting research in a previously unexamined domain is an **iterative, opportunistic, mixed-method research methodology that focuses on human subjects.** This methodology is general enough that it should be applicable to any software domain.

The methodology is applied by identifying an initial set of research questions, a set of resources that are available for pursuing the research, and a set of study designs that are appropriate for the given set of resources.

This methodology is *iterative* because of the initial conditions of the researcher. When beginning research in a new domain, it is unlikely that the initial set of research questions will fully address the issues of concern. Therefore, multiple iterations of study are necessary to evolve the research question. In addition, the methodology assumes that the researcher does not have previous experience running studies in the new domain environment, and will need to run pilot studies to learn about the constraints of the research environment and to debug issues that relate to measurement (a notoriously challenging task in software engineering research). Researchers will wish to build up sufficient domain knowledge before attempting to take advantage of valuable and rare research opportunities, such as running a case study on a software project *in vivo*. Finally, any empirical software engineering research methodology must have an iterative component, because software engineering research problems are unbounded: it is impossible to completely characterize any particular aspect of a domain because of the enormous number of context variables and the constant introduction of new technologies. As new knowledge about the domain is generated through the research, a repository which contains this knowledge must be updated. This repository will grow over time, and must maintain metadata about the origin of the knowledge, so that stakeholders can identify the sources of evidence and make judgments about confidence.

The methodology is *opportunistic* because resources to run software engineering studies are generally scarce. It is not possible to randomly sample from the population of programmers or projects to run experiments. This problem is compounded when doing research in domains with smaller programmer populations, such as high-end computing. In addition, running case studies on projects is also difficult because projects of interest are often inaccessible to the researcher. Commercial organizations are reluctant to publicize the inner workings of their software projects, and national security issues prevent this type of research for many government projects of interest. Therefore, the research must be driven by what studies *can* be performed, rather than what studies would be ideal.

The methodology employs a *mixed-methods* approach, using quantitative and qualitative research methods. Since it is assumed that quantitative studies such as controlled experiments or correlational studies cannot be done at the level of "real" projects, at least in the early stages of the research, the methodology must necessarily combine methods. The challenge of employing a mixed-method approach is synthesizing the results of the multiple types of studies. This requires the development of a repository which can incorporate the results of both types of studies.

Finally, this methodology focuses specifically on conducting *human-subject* research. A central assumption of this dissertation is that since software engineering is a fundamentally human-centric activity, any software engineering research activity must necessarily involve humans as subjects.

A basic outline of the research the methodology follows:

Before beginning the studies:

1. Identify the stakeholders
2. Identify potential subjects

Running studies over iterations (3 & 4 are run in parallel)

3. Quantitative (classroom) studies
    a. Early iterations
        - Understand constraints of the research environment
        - Validate proposed measures
    b. Middle iterations
        - Introduce modifications to improve the study
        - Develop an experimental package
    c. Later iterations
        - Hand off package to domain researcher to run studies independently
4. Qualitative studies
    a. Early iterations
        - Gather folklore
    b. Middle iterations
        - Continue to gather folklore
        - Run low impact case-studies

c. Later iterations

    - Run in-depth case studies

To carry out the methodology, the researcher requires:

- An *experience base* for storing the results of the research
- *Human resources* for carrying out the research

Each step is described in more detail in the remainder of the chapter along with a description of how we applied it to high-end computing.

3.3 Identifying the stakeholders

The consumers of the research should be identified in advance, as the needs of different stakeholders may drive the direction of the research. Examples of stakeholders are:

- Researchers
- Professors
- Programmers
- Technologists

The methodology is designed to support application by multiple researchers simultaneously. Therefore, researchers should be considered as consumers of the knowledge and not simply producers, as methodological knowledge about carrying out research in the domain is constructed over time.

Examples from our research

In our domain, professors are educating future scientific programmers and will (we hope) wish to maximize the productivity of their students. Programmers are the ones who are writing the software on the HEC systems, and are concerned with maximizing their own productivity. Technologists are developing tools to assist the programmers, and wish to maximize the productivity of their customers. We also have an additional stakeholder, not mentioned above, an administrator. Administrators purchase and maintain HEC systems and wish to maximize the average productivity of the users as well as keeping costs to a minimum.

3.4 Identifying potential subjects

A set of potential subjects of the empirical studies should be identified in advance, based on the resources that the researcher has access to. This set of subjects includes both individuals and projects. Potential candidates include:

- University courses
- Practitioners
- Publicly accessible software projects
- Individual programmers with various levels of experience in the domain

3.4.1 University courses

Quantitative studies such as controlled experiments and quasi-experiments can be integrated into a classroom environment. Case studies can also be run in a classroom environment when the class has a project component. Using students in software engineering research is a generally accepted practice; a recent survey found that 87% of subjects in published software engineering experiments were students [Sjoberg05].

The advantages of using a classroom environment are:

- Access to many subjects with some experience in the domain
- Low cost from an experimenter point of view
- Low cost from the subject point of view

It is much easier to obtain multiple subjects in a classroom environment than if the subjects need to be recruited individually. Since the research is domain-specific, the course used for conducting the research should pertain to that domain.

The main disadvantages are:

- Experience of subjects
- Small problem sizes
- Balancing pedagogy and research

Generally, students (even graduate students) do not have the same level of experience as domain practitioners, and that may introduce external validity issues that must be addressed by the researcher (although the results may still apply to the population of "novices"). In addition, there are constraints on what sorts of problems can be used for studies in a classroom. The typical programming assignment is due two to three weeks after it has been assigned, which constrains the size of the problem that can be used. Finally, there are constraints on the modifications that can be made to the assignments, as they must retain their pedagogical value and they cannot provide one group of students in the class with an unfair advantage.

Examples from our research

In our research, the domain was high-end computing, so we took advantage of courses that dealt with high-end computing and parallel algorithms. The subjects are provided with the relevant training in the domain as part of the course, so the experimenter does not need to provide the subjects with additional training. If the studies are properly integrated into the courses, then the cost of participation from the subject's point of view is minimal. The studies we ran in the classroom were always based on assignments that the students were required to do, so the only additional activities on their part were filling out some forms.

Through the HPCS project, we had the opportunity to collaborate with professors at different universities across the U.S. who were teaching courses related to parallel programming and high-end computing. These professors were willing to participate in studies, but had never before done human-subject research. They often provided us with feedback on the studies and the goals. In addition, through the HPCS project the professors were able to obtain access to certain HEC machines. Through collaboration with different labs, we were able to obtain access to the following machines:

- San Diego Supercomputing Center (SDSC)
    - IBM SP (Blue Horizon)
    - IBM p655+/p690 (Datastar)
- Army High Performance Computing Research Center (AHPCRC)
    - Cray X1 (mf)
- Arctic Region Supercomputing Center (ARSC)
    - Cray X1 (Klondike)
- Oak Ridge National Lab
    - IBM Power4 (Cheetah)

3.4.2 Practitioners

Practitioners in the domain of interest may have years or decades of experience, and the challenge for the empirical software engineering researcher is to somehow capture those elements of experience that are relevant to the research questions.

Examples from our research

Because of the high profile of the HPCS project in the HEC community, many practitioners were involved with the project. They were generally very willing to discuss their past experiences in software development. Indeed, we found that practitioners had very strong opinions about software development issues, and were very willing to voice them when provided an opportunity.

3.4.3 Publicly accessible software projects

While industry and government software projects often cannot be examined because of secrecy concerns, there are software projects that do not have such concerns and can therefore be studied more easily. Two prominent classes of projects which do not have such restrictions are

- Academic projects
- Open-source projects

Note that the development process for these types of projects may be very different from government or industry development processes, raising issues of external validity.

Examples from our research

In our case, we focused our efforts on research projects that were being conducted in academia, and we selected projects which were believed to be similar to certain government projects that we could not obtain access to. Because much of computational science today relies on HEC, there are many projects in academia that involve developing software for these systems which have no institutional barriers preventing us from interacting with project members.

Through HPCS, we had the opportunity to collaborate with all five of the ASC-Alliance centers, which are computational science research centers located at various universities in the U.S. who are provided with access to unclassified HEC machines, which are located at various NNSA labs. The five centers are:

- Stanford University Center for Integrated Turbulence Simulations
- California Institute of Technology Center for Simulation of Dynamic Response of Materials
- University of Chicago Center for Astrophysical Thermonuclear Flashes
- University of Utah Center for the Simulation of Accidental Fires & Explosions
- University of Illinois Urbana-Champaign Center for Simulation of Advanced Rockets

3.4.4 Individual programmers

Individual programmers can be used for running observational studies or for evaluating different aspects of a study design (e.g. for doing user testing of software tools used for data collection in the study). For certain types of studies, it will be important that the programmer has the appropriate domain experience (e.g. for trying out a new programming assignment to be used in a study). For other types, it is less important (e.g. validating effort measurement). Obtaining access to individual programmers is much easier than obtaining access to groups of programmers. The researcher should be able to identify local programmers to participate in such small-scale studies.

Examples from our research

Since we are in a university setting, we had access to several programmers who were willing to participate in small-scale studies. Some of these had a fair amount of experience in this domain (e.g. staff programmers, graduate students specializing in these areas), although most potential subjects were graduate students.

3.5 Iterations

One underlying assumption of this methodology is that knowledge-building in a domain is an iterative process, and is best achieved by conducting a series of studies which build on previous ones. As each research iteration is conducted, the researcher will build knowledge not only of the domain, but of methodological concerns as well, which will affect subsequent iterations.

One primary aim of an iterative process is to evolve a set of **hypotheses** over time. Because software engineering is fundamentally a human-centric process, and because of the complexity of human behavior, we do not attempt to build comprehensive theories, aspects of which can be tested, as would be the case in the natural sciences. Instead, we adopt the social science approach of generating hypotheses based on our experiences to date, which are then tested through empirical study [Vegas02].

One particular aspect of hypothesis evolution is that of understanding the role of **context variables**. In software engineering, there are a huge number of variables that can potentially affect outcomes. A major aspect of such research is simply to identify the context variables that affect outcomes. As iterations progress, the impact of an initial set of context variables can be tested, and new ones will be discovered, to be tested in later iterations.

Along with the hypotheses, even the **goals** of the research may evolve over the iterations. It is assumed that the researcher has some initial research goals before beginning to conduct studies. However, as new phenomena about the domain are discovered, the research goals may change over time. This necessitates an iterative approach, because it is impossible to plan out all of the studies necessary in advance. For example, in our particular instantiation of the methodology, the initial research goal was to understand the effect of parallel programming model on productivity. However, as the research progressed the goals expanded to include the investigation of other factors that affect productivity.

Through the iterations, the researcher will undergo a learning process about how to better carry out the studies in the environment. In the early studies, the data collection methods may change as the researcher determines what instrumentation works best. Eventually, the researcher will stabilize on a particular instrument, and will have accumulated a large body of data. As this is happening, the researcher should be developing infrastructure to manage all of the data. The requirements for this infrastructure will become clear after several iterations.

No individual study can ever be perfect, and each method of empirical study has its own weaknesses. However, all of the **weaknesses** of a particular study design may not become apparent into the study is actually conducted. By using an iterative approach, researchers gain experience with a particular type of study, and can compensate by modifying the design for the next iteration, or using a completely different study type. In this way, the quality of the studies becomes improved over time. After a certain number of iterations, one particular type of study may reach the limit of effectiveness. For example, running studies in a classroom may be a useful way to identify measurement issues and test some hypotheses *in vitro*. However, given the low levels of external validity in classroom studies, there may reach a point where this particular approach is exhausted.

Another key aspect of the iterative approach is that of **combining results across multiple studies**. Unlike the meta-analytic approaches discussed in Chapter 2, using this methodology the researcher has a measure of control over all of the studies whose results are to be combined. A new hypothesis generated from one type of study may be tested in a new study in a subsequent iteration. The weaknesses of study designs can be counterbalanced (e.g. observing a phenomenon in a controlled experiment, and verifying it occurs *in vivo* with a case study). Therefore, the researcher designs future studies to test new hypotheses and balance weaknesses of previous studies.

Below we discuss in detail how individual study designs may evolve across multiple iterations.

3.6 Quantitative (classroom) studies

Introducing an empirical study to a classroom environment where the professor teaching the course is not the same person as the empirical software engineering researcher can be challenging. The assignments used in previous classes may need to be modified to better accommodate the needs of the study. In the early iterations of the studies, we recommend against introducing changes to the course in general or to the assignments in particular. Instead, we recommend focusing on two aspects:

- Identifying the constraints of the research environment
- Identifying and validating measurement issues

3.6.1 Constraints of the research environment

The researcher will be running studies in courses where the professor teaching the course is not an empirical software engineering researcher, and does not have experience conducting human-subject research. Therefore, early studies in the classroom environment should be used to simply understand the constraints that are placed upon the researcher when conducting a study in this type of environment.

Examples from our research

We used these early studies to identify:

- What type of data we could collect automatically from the subject's environment
- What type of data we could collect directly from the subject
- Whether the kinds of data produced "naturally" by the student (e.g. writeups, code submissions) were adequate for our purposes

These early studies revealed the limits of the type of data we could collect directly from students and what types of modifications needed to be made to the assignments provided by the professor to make them more suitable for analysis in the studies.

3.6.2 Measurement

In software engineering, there are many attributes which are notoriously difficult to measure (e.g. effort, productivity, maintainability, dependability, complexity). Early iterations of the study can be used to validate these measurements and avoid problems of construct validity. Therefore, early studies may simply be exercises in measurement and instrumentation, trying to capture data on the subjects as they work through the task of interest.

Examples from our research

We used the early studies to validate the instrumentation and algorithms that we used to measure programmer effort, since our initial focus was comparing effort across programming models and problems.

3.6.3 Introducing modifications to improve the study

Once the researchers have gained some experience with data collection and measurement issues and understand the constraints of the environment, modifications can be introduced into the study, with some confidence that they will not detract from the pedagogical value of the assignments. The modifications will vary based on the maturity of the research.

Examples from our research

For example, consider the following modifications that were introduced in successive iterations in this research:

1. Modification of assignment description to standardize reporting of performance data
2. Controlling the order in which students solved a problem using multiple programming language
3. Introducing new course material to measure its effect

An early change, modifying the assignment description, was a minor issue. The later changes have larger impacts: controlling the order that students solve problems involved random assignments of subjects into treatment groups, which was additional effort for the professors, and was only done after the initial iterations. The final modification involved an actual change in the course itself, which was only attempted after many iterations of research, when we had some notion as to what educational treatment might benefit students.

3.6.4 Developing an experimental package

As the studies progress through multiple iterations, the professors will gain more experience with conducting a study (e.g. presenting the study to the students, ensuring that data collection is being performed, etc.), and the materials used to run the study will mature (e.g. assignment descriptions with more explicit requirements), until the point is reached where the experiment can be run entirely by the professor. This drastically reduces the cost of the study from the point of view of the researcher.

One important aspect of this experimental package is a repository for storing all of the raw data from the studies (this is separate from the experience base, which stores information at a higher level of abstraction).

Examples from our research

We developed a set of tools for automatically collecting data that could be deployed by the professor or TA associated with the course. We also developed a web-based Experiment Manager system for collecting and storing the relevant experimental data. In addition, we accumulated a set of classroom assignments from earlier studies which could be quickly adopted by a professor and used to run a study.

3.7 Qualitative studies

As in the case of the classroom experiments, the qualitative studies will evolve and mature over the course of iterations. It is expected that the researcher will begin the research with some initial questions in mind. These should serve as a starting point for collecting qualitative data, but it is important to avoid "tunnel vision" (committing to the initial research questions too closely).

Examples from our research

In our case, the starting point was understanding the effect of *parallel programming language* on programmer productivity. However, as the research progress the scope of the research questioned broadened to include other factors that may affect programmer productivity aside from programming language.

3.7.2 Folklore

As mentioned earlier, one of the challenges of this type of research is often identifying the right questions to ask. This methodology employs a "folklore gathering" approach, to elicit tacit knowledge from practitioners. This activity should begin as early in the research as possible and should persist across multiple iterations as new issues are discovered.

Once the researchers have analyzed the folklore from early iterations, they should have sufficient familiarity with the domain and the language of practitioners to perform more in-depth qualitative studies such as individual interviews and case studies. Because practitioners can be difficult to gain access to, researchers should be well-prepared before they approach practitioners one-on-one (In a group interview, there is less concern about lack of familiarity because the researcher is observing and directing a discussion rather than directly interviewing).

Examples from our research

Our early attempts at collecting folklore were purposefully relatively unstructured and involved multiple practitioners, using email conversations and teleconference calls that were used as focus groups. This allowed the practitioners to play off of each other's responses and provided us with a wealth of qualitative information.

3.7.3 Case studies

Case studies are a useful technique for studying software projects *in vivo*. The process of conducting a case study should itself be iterative. The researcher should begin by being as unobtrusive as possible, initially making very small demands on the subjects and then increasing the level of demands placed on the subjects as confidence is built and the researcher understands more of the details of the case study.

Examples from our research

The strategy that we employed in this research for conducting case studies of the ASC-Alliance projects was as follows:

1. Brief questionnaire
2. Telephone interview with software leads
3. Instrumented or observed study of development activity[5]

In our research, case studies were not attempted until we had reached a level of maturity where in terms of the folklore gathering and the classroom assignments.

3.8 Experience base

As the studies progress, the researcher will incrementally build a *repository* or *experience base* [Basili94] that contains the results of the studies. Given that the nature of the results may vary substantially from one research area to the next, it is not possible to describe a generalized repository for containing the evidence.

[5] This is future work, planned for fall 2006.

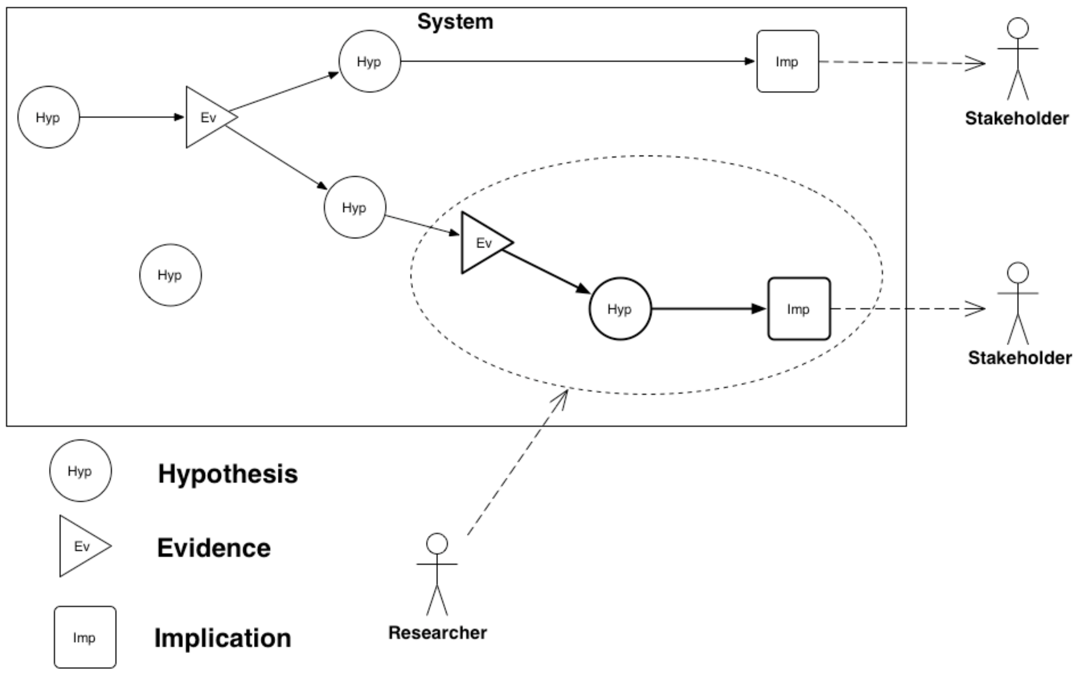


Figure 3.2 Experience base

Figure 3.2 is a diagram which shows the basic contents of the potential implementation of the experience base used in this work. The base contains hypotheses (depicted in the diagram as circles), evidence from studies (depicted as triangles) and implications of hypotheses that pertain to various stakeholders (depicted as squares). As studies are conducted, new hypotheses as generated, either through modification of previous hypotheses or from a new phenomenon observed in a study.

After a study, the researcher adds the new hypothesis that has formed, as well as a description of the evidence that has motivated the development of the new hypothesis. The researcher must also specify the implications of the hypothesis for the various stakeholders. Stakeholders interact with the system by viewing the implications, whose origins they can trace back through the existing links.

Disseminating the knowledge gained in the studies is often through the professors involved in the classroom studies. The domain knowledge gained from the studies should be fed back to the professors as soon as possible, so that they can transfer this to the students.

Examples from our research

The structure of the experience base should not vary by domain. Details of our implementation of the experience base can be found in Chapter 7.

3.9 Human resources required

The work required to implement the methodology is beyond the scope of an individual researcher. Therefore, this approach requires a team approach, with team members playing different roles.

3.9.1 Technical lead

One member of the team should be involved, at some level, with all of the studies that are being conducted and have detailed knowledge of the mechanics of the studies.

Examples from our research

In the research described here, the author played the role of technical lead. While the author did not conduct each study, he was directly involved in the majority of the studies, along with the development of many of the data collection tools. Some studies and research efforts described in this dissertation were conducted by the author, and in other studies, the author contributed in some fashion. In the initial stages of this research, Dr. Jeffrey Carver shared the technical lead responsibilities with the author.

3.9.2 Domain expert

At least one member of the research team should be an expert in the domain to compensate for the other researchers' lack of domain knowledge.

Examples from our research

In our instantiation of the methodology, Dr. Jeffrey Hollingsworth served as domain expert. Along with providing input on the design of studies and providing us with the classroom environment for running our first study, Dr. Hollingsworth generated our initial list of folklore, answered questions about domain-specific issues (e.g. terminology, whether his experience matched our results), and served as a liaison to the larger HEC community, helping us publish results in that community and putting us in touch with professors at different universities.

3.9.3 Tool development

When conducting research in a new domain, the tools required to collect, store, and analyze the data generated by the studies may not exist. The researcher must therefore build these tools, which are software projects on their own.

Examples from our research

Details of the tool development from our research can be found in Chapter 4. In the early studies, tool development work was done by the author and Dr. Jeffrey Carver. In various stages of the research, Thiago Craveiro, Patrick Borek, Nico Zazworka and

Martin Voelp worked full-time on data collection and analysis, and in later stages of the research tool development has been headed by Taiga Nakamura.

3.9.4 Documentation and artifacts

This methodology also involves the generation of documentation and artifacts which must be shared with professors who are using their classrooms to conduct research, such as experimental protocols, assignment descriptions, and documentation for the software tools.

Examples from our research

In our work, this documentation was managed by Dr. Forrest Shull and Patricia Costa. Information about Institutional Review Board approval (necessary for human-subject research at universities) was handled by Dr. Sima Asgari.

Chapter 4 Studies conducted

4.1 Introduction

In this chapter, we provide an overview of the goals and key results from the research, organized by iteration. The details of how the methodology evolved can be found in Chapter 5. A selection of results can be found in Chapter 6, and all results can be found in the online experience base described in Chapter 7. Addition details of the studies conducted in each iteration can be found in Appendix A

4.2 Iteration 1

4.2.1 Goals

- Collect effort and workflow data in a classroom environment
- Compare across programming models

We considered the first iteration to be a *pre-pilot* phase, where only one classroom study was conducted. The goal was to try and collect effort and workflow data in a classroom environment, as well as to try and compare across two programming models (MPI and OpenMP).

4.2.2 Key results

We were able to instrument the students' development environment and capture data whenever they compiled or ran their programs. We used this data to measure effort (which we refer to as *instrumented effort*). However, there were inconsistencies between effort reported by the students in a log and the instrumented effort. In some cases, the instrumented effort was very low (less than 10 minutes), which suggested that it was not an accurate measure when the student did not interact with the machine very much.

We were able to measure a statistically significant difference in effort between a programming problem the students solved using MPI and a problem they solved using OpenMP, but the problems were so different that we had very little confidence in these results.

4.3 Iteration 2

4.3.1 Goals

- Familiarize professors with running these types of studies
- Compare programming models directly, including local research languages
- Evaluate instrumentation modifications for improved effort capture

In iteration 2, we attempted to run classroom studies simultaneously in multiple courses at multiple sites. A primary goal was to familiarize the professors with the mechanics of running studies in their classes. We also attempted to compare programming models directly using the same problem, where possible. Along with MPI and OpenMP, we included research technologies of interest to the professors (MATLAB*P, XMT-C).

In addition, we had modified our data collection software to capture some additional data from the subjects, to increase the accuracy of our instrumented effort, and we used these studies to evaluate these modifications.

4.3.2 Key results

We were able to obtain statistically significant results when comparing MPI to OpenMP effort in 2 out of 4 problems where direct comparisons were possible (see Section 6.2.2 for more details). However, since we did not control for the order in which students solved the problems, we could not determine whether the effect on effort was primarily due to programming model, or due to having previously solved the problem with a different programming model.

We were not able to evaluate the research technologies. For MATLAB*P, we were not able to collect instrumentation data because our instrumentation tools do not work with an interpreted environment such as MATLAB. We could not evaluate XMT-C because the compiler was not mature enough to be used in an evaluation study: the students spent too much time working around compiler problems.

We were not able to properly evaluate our new instrumented effort measure because we did not have sufficient confidence in the reported effort provided by the students to use it as a basis for comparison.

4.4 Iteration 3

4.4.1 Goals

- Evaluate a new language (Co-Array Fortran)
- Familiarize new professors with running studies
- Capture information about defects

- Validate effort measures
- Capture folklore

In iteration 3, we attempted to evaluate a new language in a classroom environment: Co-Array Fortran, which required us to instrument a new machine: a Cray X1 system. This study was also done to familiarize the professors with running studies, since they had never run studies before in their classroom. We also ran a pilot study to determine whether we could study time spent debugging by capturing defect data.

In addition to the classroom studies, we ran observational studies to evaluate our instrumented effort measure. We also began to capture folklore through email discussions and a focus group session to gain a more qualitative understanding of development time issues.

4.4.2 Key results

Direct evaluation of Co-Array Fortran was not possible in this particular study because there were not enough data points, and the Co-Array Fortran programming assignment, while similar to the MPI and OpenMP assignments, was not identical. However, the machine was instrumented successfully, and was used in later iterations.

The pilot defect collection study showed that the forms were useful in capturing data about time spent fixing defects, and these forms were used in subsequent studies.

Through the observational studies, we were able to estimate the accuracy of our effort measure. In addition, we developed a new self-reported effort log and a measure for comparing self-reported effort data to instrumented effort.

The following elements of folklore were originally collected by Dr. Jeffrey Hollingsworth and served as the basis of the folklore focus group:

OFL.1. Use of Parallel machines is not just for more CPU power, but also for more total memory or total cache (at a given level).

OFL.2. It's hard to create a parallel language that provides good performance across multiple platforms.

OFL.3. It's easier to get something working in using a shared memory model than message passing.

OFL.4. It's harder to debug shared memory programs due to race conditions involving shared regions.

OFL.5. Explicit distributed memory programming results in programs that run faster since programmers are forced to think about data distribution (and thus locality) issues.

OFL.6. In master/worker parallelism, the master soon becomes the bottleneck and thus systems with a single master will not scale.

OFL.7. Overlapping computation and communication can result in at most a 2x speedup in a program.

OFL.8. HPF's data distribution process is also useful for SMP systems since it makes programmers think about locality issues.

These elements were refined after the focus group discussion, the refined folklore can be found in Appendix D.

4.5 Iteration 4

4.5.1 Goals

- Compare models with better controls for ordering
- Compare defects across models
- Compare across classes
- Measure folklore consensus

In iteration 4, many of the professors involved had already run studies in previous iterations, and were willing to modify their assignments to improve the validity of the study.

We used random assignment into treatment groups, but the nature of this process varied by class. In one class, the professor was very ambitious and wanted to use four different programming models: MPI, OpenMP, UPC/Co-Array Fortran, and MATLAB*P. Students were assigned one model, and chose one other model. In other classes, the controls were simpler. In one class, the students solved a problem in both MPI and OpenMP, but half solved it first with MPI, and the other half solved it first for OpenMP. In another class, half the students solved a problem using only MPI, and the other half using only OpenMP: in the latter class, we collected defect data and compared the time to fix MPI defects with the time to fix OpenMP defects. In yet another class, the professor teaching his class only XMT-C chose an MPI assignment used in a class from iteration 2, so comparisons could be done without teaching the students a second programming model.

4.5.2 Key results

The controls for ordering improved the validity of the studies. However, one of the controls (assigning one model and choosing another) proved to be too complex for us to analyze. It reduced the number of data points for comparing between models, and it did not help in controlling for ordering. The XMT-C/MPI effort comparison did provide statistically significant results, but the other comparisons did not.

The comparison of defects between MPI and OpenMP did not yield statistically significant results, which contradicted the hypothesis that shared memory programs are harder to debug.

4.6 Iteration 5

4.6.1Goals

- More in-depth folklore research
- Evaluate effort measure in more natural work environment

The folklore gathered in the previous iterations were transformed into an interview guide, so that we could collect more detailed, qualitative information about the folklore from practitioners.

We ran a third observational study in a much more natural work environment for the subject than the previous studies, to minimize the Hawthorne effect and determine whether this would have an impact on the accuracy of the effort measurement.

4.6.2 Key results

The observational study revealed that error in the instrumented effort increased substantially when the subject's work environment during the study was more similar to their typical work environment.

The interviews revealed some differences in opinion about folklore among practitioners. For example, when asked about race conditions, one mentioned that no race conditions occur in message-passing, only in shared memory, and another responded that anything in shared in memory is easier to debug than in message-passing. Another example was the scalability of OpenMP: one practitioner claimed to have never seen OpenMP used seriously on more than 4 processors, another one mentioned having seen it scale to 64 processors or better.

4.7 Iteration 6

4.7.1 Goals

- Compare models with good control for ordering
- Examine larger student projects
- Examine realistic sized projects (ASC-Alliance projects)

In iteration 6, we were able to improve the controls of the study to enforce the ordering constraint: half of the students solved the problem in MPI, half in OpenMP, then they turned the assignment in, and then they solved the problem again with the other model.

We had several research questions in mind when conducting case studies with the class projects. We wanted to understand where the students spent their time on a project and contrast that with the classroom assignments to understand how the distribution of time changes with scale. We wished to know if relatively inexperienced parallel programmers could actually achieve reasonable performance. Finally, we wanted to compare the results across case studies to get some sense of the effect of problem type on effort.

In addition to the case studies conducted in the classroom, in this iteration we began to move towards case studies of actual projects. The goals here were to understand much more about programming issues "in-the-large" and try to characterize where developers on larger HEC projects spent their time in the different development activities. We wanted to characterize the differences between programming in classroom assignments and programming in "real" projects.

4.7.2 Key results

In the classroom study, we obtained statistically significant results when comparing MPI to OpenMP, as well as measuring the effect of having solved the problem before in a previous model (see Section 6.3.3).

The class project case studies revealed that the students were able to obtain reasonable performance, despite essentially no performance tuning (see Section 6.5.2 for further discussion). They generally got better performance improvements than they did on the class assignment they solved earlier in the semester, which illustrated that problem size is not necessarily related to the difficulty in achieving speedup. In addition, for the two projects that were porting a pre-existing (unfamiliar) MATLAB code to C or C++, the time to port to C/C++ outweighed the time spent parallelizing.

Studying the ASC-Alliance projects revealed a number of issues that were different from our experiences in other domains, such as:

- Most of the programmers were scientists without computer science training
- Most of the users were programmers who working on the code
- Almost no time was spent on tuning for a particular architecture
- They drew a distinction between finding defects in the algorithm they were evaluating, and defects in their implementation
- All projects used MPI exclusively for parallelism, although some had built layers on top of MPI to provide a higher level of abstraction
- Small but consistent set of software engineering practices were used: for example, all used version control, but almost none used any bug tracking, and there was almost no design documentation

Chapter 5 Evolving the methodological infrastructure

5.1 Introduction

This chapter is organized by the types of studies conducted, and describes how the methodology of the studies mentioned in the previous chapter evolved across the iterations.

5.2 Research activities

The empirical work conducted in this dissertation can be divided up into the following five categories.

- Classroom studies
- Observational studies
- Folklore gathering
- ASC-Alliance case studies
- Classroom case studies

5.3 Classroom studies

With access to university courses, we could run studies that involved multiple subjects. However, since we did not have existing tools or methodologies for running studies in this environment, we needed to develop the experimental design, artifacts, data collection, and analysis tools ourselves. We used these studies to prototype our data collection and analysis methods, to compare programming models directly, and to try to collect some data on where programmers spend their time when writing HEC programs. Classrooms studies were run in each iteration except for iteration 5 (see Appendix A).

5.3.1 Experimental design

The experimental design of the classroom studies evolved as we gained experience running studies. In addition, the constraints on the studies varied from class to class, and so the experimental designs had to be adopted accordingly. Because the studies took place in a classroom environment, and the fundamental goal of the assignments was pedagogical, there were limits on the type of constraints we could put on these studies. In addition, there were sometimes limits on the type of data we could collect from the environment.

The initial classroom studies involved simply collecting data on the students as they worked, and we did not attempt to interfere with the normal classroom assignment. The initial attempts were to gain experience with the process of running studies in this environment, for both the experimenters and the professors involved in teaching the courses.

After we gained some experience with running studies, we noticed a particular threat to validity when an assignment involved multiple programming models (e.g. students required to solve the game of life in MPI and OpenMP). There was no restriction on the order in which students solved the problem: if all students solved the problem in, say, OpenMP first, and then MPI, and the result of the study was that OpenMP required less effort, we could not factor out the effect of having solved the problem using the previous model.

In later studies, we controlled for ordering in various ways. In one study, half of the students solved the problem first using one model, and half solved the problem using another model, and then they switched, solving the same problem. In this way, within-subject comparisons were possible. In another study, comparisons were made across two completely separate classes: in one class, students solved a problem with only one model, and in another class, students solved a problem with another model.

The initial studies focused only on measuring the total effort involved and comparing across programming model and problem. In later studies, with some additional controls we were able to test other hypotheses. For example, we ran studies to compare time-to-fix defects across programming models, or to measure the effect of experience with a problem, or the effect of tool use[6].

5.3.2 Experimental artifacts

In the classroom studies, the primary artifacts are the assignment descriptions. In most of the assignments, students are only given a description of the problem to be solved. However, in some cases they are also provided with sample inputs, and in other cases they are provided with source code for a sequential implementation.

As with the experimental design, in early studies there were no attempts to influence the assignment process. However, after conducting the initial studies we discovered that there were certain ambiguities in the assignment descriptions that made it more difficult to analyze the resulting data. The assignments were often vague about what sort of performance data should be reported by the students (e.g.

[6] The tool use study occurred after the studies described in this dissertation.

inputs to be used, number of processors). Therefore, the resulting data across subjects was difficult to compare. In addition, the assignments were often not specific about how important it was to achieve good performance. We were therefore concerned that each subject would have a different interpretation of the level of performance they were expected to achieve, and this would affect their effort. To avoid this problem, we developed a template that was to be used by the professors to make these issues more explicit and achieve consistency across studies. This template can be found in Appendix K.

5.3.3 Data collection forms

Several data collection forms were developed and evolved over the course of the study.

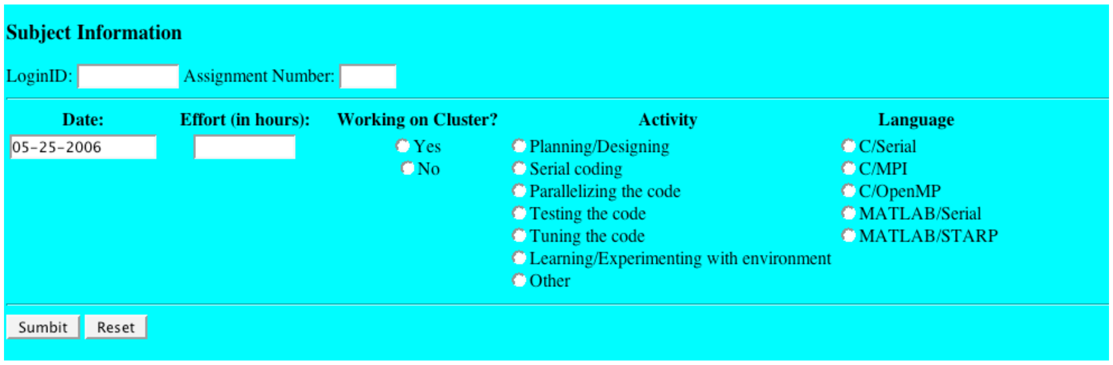
Effort forms

We developed effort logs to be used for subjects to keep track of their own effort. These logs underwent considerable evolution throughout the course of the research.

Effort (hours)	Thinking (Understanding the problem)	Thinking (Designing a solution)	Experimenting with environment	Adding functionality	Parallelizing	Tuning	Debugging	Testing	Other (specify activity)
0.25	X	X							

Figure 5.1 Initial effort log

Figure 5.1 shows the formatting of the initial log used in iteration 1. This was a pen-and-paper log, which subjects used to report how much time they spent on a particular task, and classified it based on a set of categories (see Section 5.3.6 for details on the evolution of these categories).


Subject Information

LoginID: Assignment Number:

Date:	Effort (in hours):	Working on Cluster?	Activity	Language
05-25-2006		○ Yes ○ No	○ Planning/Designing ○ Serial coding ○ Parallelizing the code ○ Testing the code ○ Tuning the code ○ Learning/Experimenting with environment ○ Other	○ C/Serial ○ C/MPI ○ C/OpenMP ○ MATLAB/Serial ○ MATLAB/STARP

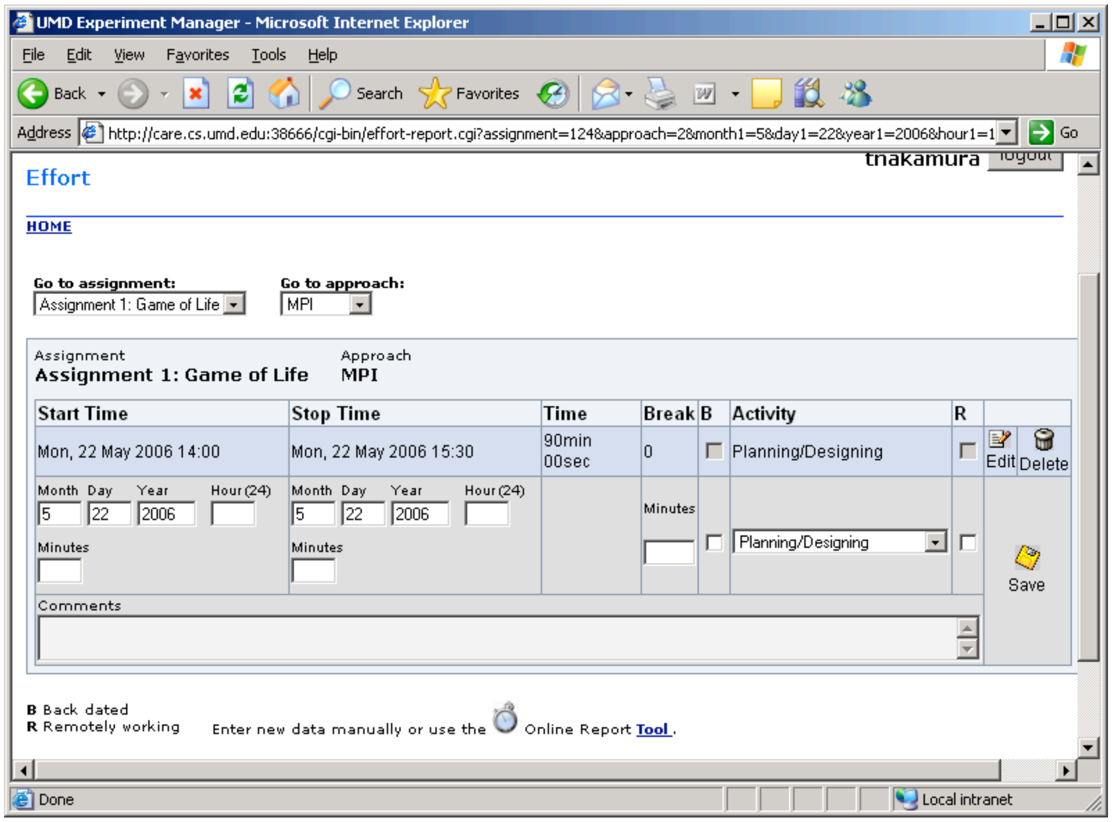
Figure 5.2 Initial web-based effort log

Figure 5.2 shows the second incarnation of the effort log, introduced in iteration 2, which had been changed to a web-based log to simplify the data collection process (no transcribing of logs necessary), with the hopes that it would improve reporting rates (it did not). This log was augmented with information about the date, and a question about whether the subject was working on the "cluster" (the HEC machine provided to the students by the instructor). This information was added to facilitate cross-checking this log with the automatically collected data (discussed later).

Date	Working remotely	Start time	Stop time	Breaks (minutes)	Total time (minutes)	Language/ approach	Activity	Comments	Back dated
9/29		10:35	11:45	5	70	Fortran/Serial	Th		
"		11:45	12:45		60	"	Se		
"		19:00	20:30		30	"	De		

Figure 5.3 Second paper-based effort log

Figure 5.3 shows the third incarnation of the effort log, originally developed for use in Obs-2 of iteration 3. The subject specifies the activity using a two-character code (Th=thinking, Se=serial coding, Pa=Parallelizing the code, Te=Testing, De=Debugging, Tu=Tuning, Ex=Experimenting with environment, Ot=Other), We switched back to paper because one of the perceived benefits of a web-based log (better subject compliance) did not appear in the studies and because it is easier to modify a paper-based log than a web-based log when experimenting with log formats. The third incarnation of the log has two modifications over the previous version. The main difference is how effort is reported: subjects report start/stop times instead of reporting their effort in hours. The reason for this modification was to further simplify the task of comparing the self-reported data to data that was automatically collected. The other modification was to add the "backdated" field, which indicates whether the subject filled out the log as they were working, or filled it out retrospectively ("backdated" the log). We are interested in capturing this because retrospective logs are known to be less accurate [Perry96].


UMD Experiment Manager - Microsoft Internet Explorer

Address http://care.cs.umd.edu:38666/cgi-bin/effort-report.cgi?assignment=124&approach=2&month1=5&day1=22&year1=2006&hour1=1

tnakamura logout

Effort

HOME

Go to assignment: Assignment 1: Game of Life

Go to approach: MPI

Assignment: Assignment 1: Game of Life

Approach: MPI

Start Time	Stop Time	Time	Break	B	Activity	R	
Mon, 22 May 2006 14:00	Mon, 22 May 2006 15:30	90min 00sec	0	☐	Planning/Designing	☐	Edit Delete
Month 5 Day 22 Year 2006 Hour(24) Minutes	Month 5 Day 22 Year 2006 Hour(24) Minutes		Minutes	☐	Planning/Designing	☐	Save

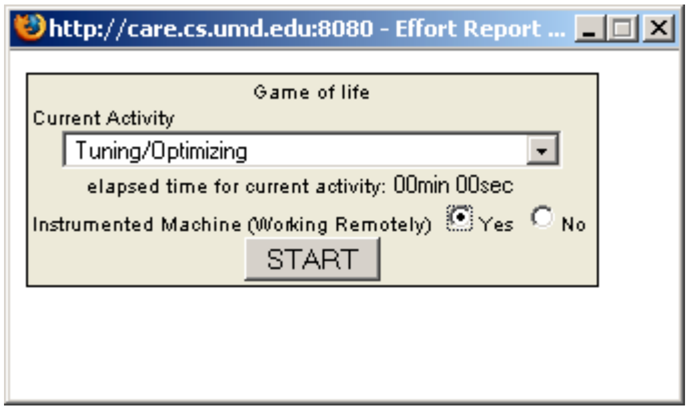
Comments

B Back dated
R Remotely working

Enter new data manually or use the Online Report Tool.

Done Local intranet

Figure 5.4 Second web-based effort log


http://care.cs.umd.edu:8080 - Effort Report ...

Game of life

Current Activity: Tuning/Optimizing

elapsed time for current activity: 00min 00sec

Instrumented Machine (Working Remotely) [✓ Yes] No

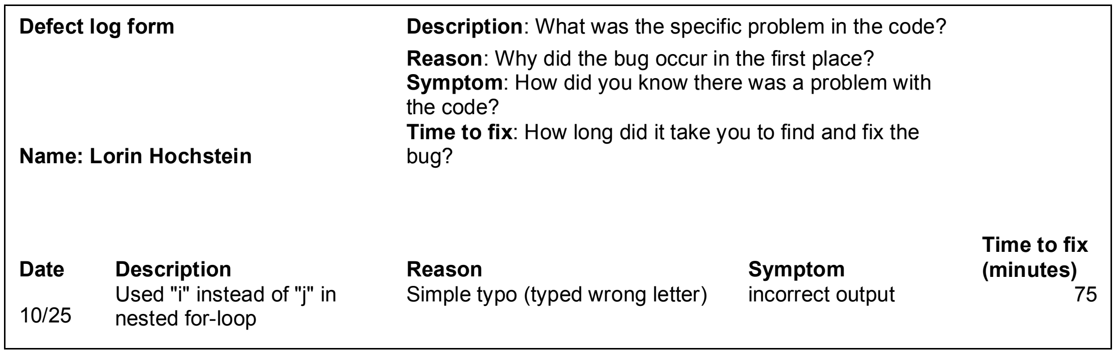
START

Figure 5.5 Effort report tool

Figure 5.4 shows the fourth and current incarnation of the effort log. Since we were essentially satisfied with the format of the paper-based log, it was implemented again as a web-based interface so that the data was stored automatically in a database when submitted by subjects. The log supports two types of data entry: one type is identical to the previous form, where the user specifies start/stop times. The other type is a “stopwatch” interface where the user indicates when work is beginning and when work is ending, so that the user does not have to track start/stop times manually (see Figure 5.5).

Defect forms

In later studies, defect logs were introduced where the focus of the study was specifically on types of defects and time spent debugging. The log used is shown in Figure 5.6.



Defect log form

Name: Lorin Hochstein

Description: What was the specific problem in the code?
Reason: Why did the bug occur in the first place?
Symptom: How did you know there was a problem with the code?
Time to fix: How long did it take you to find and fix the bug?

Date	Description	Reason	Symptom	Time to fix (minutes)
10/25	Used "i" instead of "j" in nested for-loop	Simple typo (typed wrong letter)	incorrect output	75

Figure 5.6 Defect log

The defect log captures the notions of fault/failure/error associated with defects as specified by the IEEE Standard [IEEE-610.12], but it uses different terminology (see Table 5.1). The log also captures date and time-to-fix information. The date is used to cross-check against other data that is collected, and the time-to-fix information is used for quantitative analysis.

Table 5.1 Defect terminology

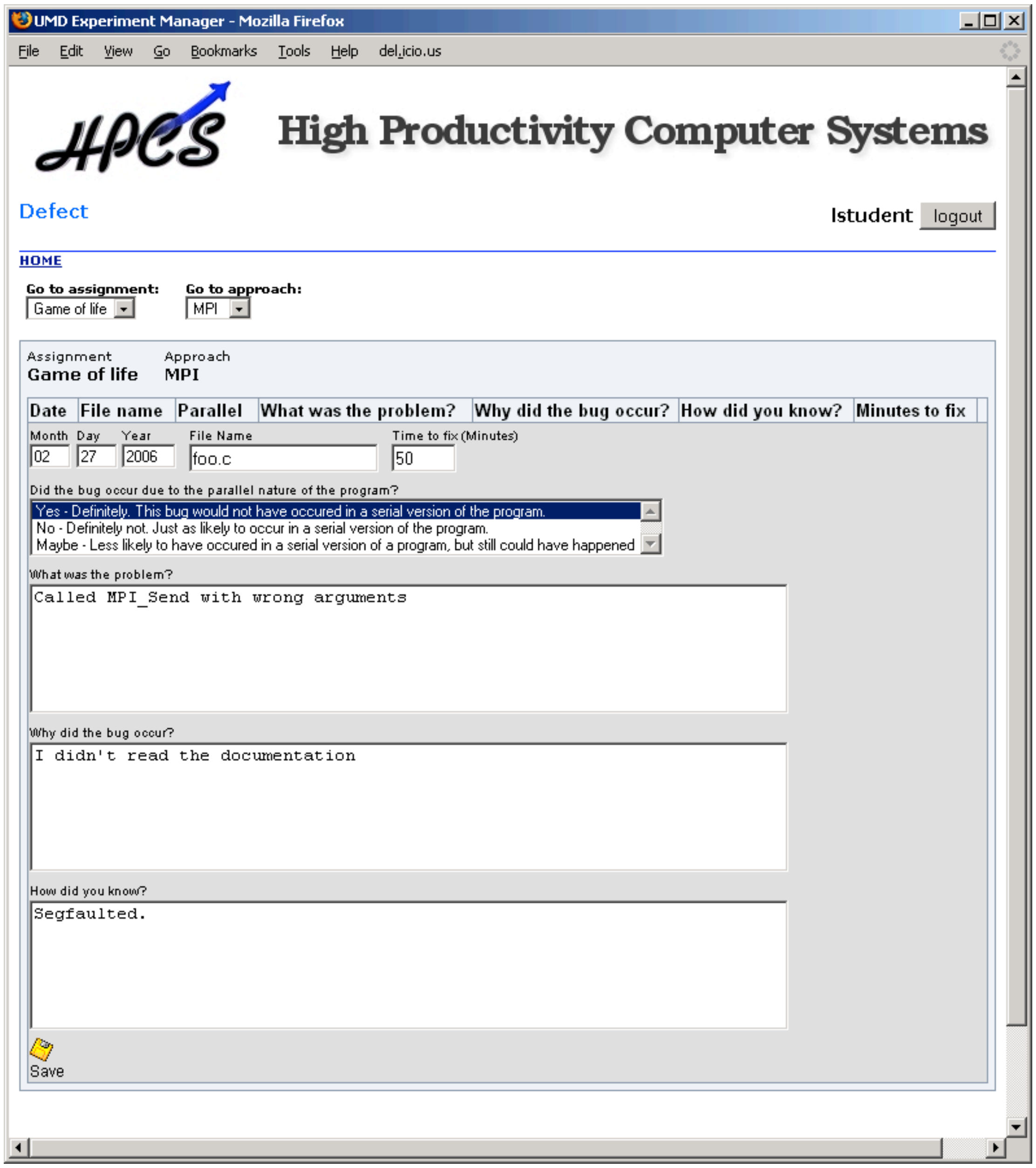
IEEE Standard	**Defect paper log**	**Defect web log**
Fault	Description	What was the problem?
Failure	Symptom	How did you know?
Error	Reason	Why did the bug occur?

These forms have been converted to web-based forms as well, so that the data is stored directly into a database when it is captured (see Figure 5.7).

5.3.4 Automatic instrumentation

In addition to the data collection forms described in the previous section, we captured data automatically while the subjects worked. This automatic data collection was largely unobtrusive, although in some studies the instrumentation software asked the subjects questions while they worked (see Section 5.3.6). The instrumentation evolved over the course of the studies.

Since the subjects in the classroom studies typically did not have access to their own parallel machine, they compiled and ran their programs on a parallel machine provided to them by the university. This allowed us to install instrumentation on a single machine, rather than on each subject's individual machine.


UMD Experiment Manager - Mozilla Firefox
File Edit View Go Bookmarks Tools Help del.icio.us
HPCS
High Productivity Computer Systems
Defect
lstudent
logout
HOME

Go to assignment: Game of life
Go to approach: MPI

Assignment: Game of life
Approach: MPI

Date	File name	Parallel	What was the problem?	Why did the bug occur?	How did you know?	Minutes to fix

Month: 02 Day: 27 Year: 2006
File Name: foo.c
Time to fix (Minutes): 50

Did the bug occur due to the parallel nature of the program?
[✓ Yes - Definitely. This bug would not have occured in a serial version of the program.]
No - Definitely not. Just as likely to occur in a serial version of the program.
Maybe - Less likely to have occured in a serial version of a program, but still could have happened

What was the problem?
```
Called MPI_Send with wrong arguments
```
Why did the bug occur?
```
I didn't read the documentation
```
How did you know?
```
Segfaulted.
```
Save

Figure 5.7 Web-based defect report form

Wrappers

At the beginning of this research, we were not able to locate any existing data collection tools that would suit our purposes, so we developed our own. The primary tools that we developed are “wrappers”, programs that are designed to “wrap” an existing program and capture some additional information. From the subject's point of view, using these data collection tools were transparent: they simply invoked the "wrapped" programs as they would invoke them normally.

The wrappers were primarily used for instrumenting compilers. In the initial version of the instrumentation, each time a subject compiled a program, the instrumentation would capture:

- A timestamp
- Contents of the source files that were compiled
- The command used to invoke the compiler

Later versions of the instrumentation also captured:

- The return code of the compiler (indicates success/failure)
- The time to compile
- Contents of local header files referenced in the source files

The compiler wrappers were also designed so that they could ask the user questions for each compile. These questions were used to ask subjects what they were doing and also to ask him them how long they had been working since the previous compile (see Section 5.3.5 for more details).

The wrappers were also used for wrapping programs other than compilers. Initially, they were used to capture data from the program used to submit jobs to the machine (many HEC machines are batch scheduled, and programmers must submit a job to a batch queue). In later studies, the wrappers were augmented to also wrap programs that run jobs interactively (e.g. "mpirun"), as well as debuggers and profilers.

Editor sensors

Beginning in iteration 2, we also used the Hackystat framework [Johnson03] for collecting additional data during development. The Hackystat framework uses sensors that plug into different software development tools, capture data, and transmit the data to a central repository. Hackystat was not explicitly designed for conducting these types of experiments, but we were able to adopt some of the Hackystat functionality for our own purposes.

The first Hackystat sensors we incorporated into our studies were the editor sensors. These sensors captured timestamped events such as when files are opened/closed/saved, when the user switches buffers, and when a buffer is modified. We used only the Emacs editor and vi editor sensors, since these were the most common editors that were both available on the HEC machines used in the studies and were supported by Hackystat.

Since Hackystat depends upon the Java runtime environment, we were not able to use it on machines that did not have Java installed. The only such machines we encountered were Cray X1 systems.

Shell capture

Starting in iteration 3, we also captured the shell commands issued by the user. This was captured using the Hackystat CLI sensor when it was available (i.e. on machines with Java). On machines where it was not available, we wrote our own software to capture this information.

We were only able to capture this information when the subject used tcsh as their shell (because tcsh keeps a timestamped history), so the later studies required that we ensure that subjects used tcsh as their shell.

5.3.5 Effort measurement

One of our primary goals in the classroom studies was to measure and compare the effort across programming models and problems. We used the effort forms and data collection tools described in the previous sections to estimate effort. Obtaining effort estimates from the self-reported effort forms was straightforward, but response rates for the effort forms were quite low and their accuracy was uncertain, so we could not rely solely on these forms.

The automatically collected data did not provide us directly with effort estimates, but instead with streams of timestamped events, which had to be processed to form an effort estimate. The algorithm used to compute the effort varied over the course of the studies, as described below.

After the initial study (A-F03), we used a simple algorithm for estimating effort based on the timestamps from the compiles and runs. We computed the time interval between successive events. If this interval was less than 45 minutes, then we counted the entire interval as effort expended by the subject. If the interval exceeded 45 minutes, we assumed the student did some other non-project activity and we counted 45 minutes as the effort for the interval as the average amount of time spent thinking about the problem during the elapsed interval. Formally, we can write these equations as:

$$E = \sum_{i} f(t_i)$$

where E is total effort, t_i is the i'th time interval (between compiles *i-1* and *i*), and f is the following function:

$$f(t) = \begin{cases} t & t \le T_1 \\ T_2 & t > T_1 \end{cases}$$

where we initially chose $T_1=T_2= 45$ minutes.

After the initial study, we found poor agreement between the reported effort and the instrumented effort. We then modified the instrumentation to try to more accurately capture the time that the subject spent working between compiles. The interface appears in Figure 5.8.

Figure 5.8 Instrumentation querying user for time spent working

```
$ mpicc life.c

How long (in minutes) have you been working before this compile?
(Hit enter if you have been working continuously since last compile)
>
```

This new data required that we modify our algorithm:

$$E = \sum_i f(t_i, w_i)$$

where w_i is the "work time" specified by the user (0 by default), and f is the following function:

$$f(t_i, w_i) = \begin{cases} w_i & w_i > 0 \\ t_i & w_i = 0 \wedge t_i \le T_1 \\ \overline{w} & w_i = 0 \wedge t_i > T_1 \wedge \overline{w} > 0 \\ T_2 & w_i = 0 \wedge t_i > T_1 \wedge \overline{w} = 0 \end{cases}$$

In this equation, $\overline{w}$ is the average work time specified by the subject across all compiles. If the subject specifies a work time, we use that as the time interval. If the subject does not specify a work time, we use the actual time interval, provided it falls below a threshold T_1. If the user does not specify a time interval and the actual time interval exceeds T_1, we use the mean work time specified by the subject, as an estimate of the time interval. If the subject never specified a work time throughout the development process, we use the value T_2. We determined T_1 and T_2 from analysis of the collected data from studies in iteration 2 (see [Hochstein05] for more details).

Even with the modified algorithm, we still saw significant variation between the estimated effort data and the self-reported effort data. This motivated the observational studies (see Section 5.4) to evaluate the automated effort algorithm with a more reliable evaluation mechanism then self-reported effort data, and modifications to the effort log (see Section 5.3.5).

In the end, the studies indicated that the "work time" provided by subjects was not reliable, and with the additional data gained by instrumenting editors and the shell in the later studies (see Section 5.3.4), the simple algorithm that we began with provided better accuracy than incorporating the "work time" data in the observational studies, with the modification that T_1=45 minute, T_2=0min.

To obtain a measure of total effort, we combine instrumented and self-reported effort measures using the following equation:

$$E_{total} = E_{inst} + (1-k)E_{rep}$$

where E_{total} is total effort, E_{inst} is instrumented effort, E_{rep} is self-reported effort, and k

is the fraction of the self-reported effort that corresponds to work done on an instrumented machine.

We use *fidelity* as a confidence measure to help us judge whether we should use the self-reported effort or discard it, based on how well it agrees with the instrumented effort:

$$f = \frac{Ov(E_{inst}, kE_{rep})}{E_{inst}}$$

where f is fidelity, and Ov is the overlap between two effort measures. Further details can be found in [Hochstein05].

5.3.6 Workflow measurement

In the previous section, we discussed the methodology used to estimate the amount of effort that a subject spent trying to solve a parallel programming problem. However, to gain a deeper understanding of the software development process, we would like to know how this time is spent. We use the term "workflow" to capture the time that the subject spent in different development activities, along with the order of these activities. Note that we are interested in activities at a fine level of granularity, since the programming tasks that are under investigation typically take on the order of hours rather than weeks, so we expect that the time between activity transitions to be on the order of minutes or hours.

Accurately measuring development activity at this level of granularity is still an open question in software engineering. In this section, we describe the methodology that we used to capture workflow and how this methodology evolved over time.

Broadly speaking, the methodology evolved from trying to elicit as much information directly from the subject as possible, to trying to minimize disturbing the subject. To accomplish this, we used both the self-reported effort logs and the automatic instrumentation.

The effort logs were a natural fit for collecting workflow data: rather than simply ask the subjects when they worked, we also asked them to tell us what they were doing. We also took advantage of the instrumentation to elicit information from the subjects while they were developing their software. Whenever the subjects compiled, a menu would pop up that would ask them to select from a list of activities to describe what they were doing.

Table 5.2 Evolution of activities

Iteration	Effort log activities	Compiler activities
1	1. Thinking (understanding the problem) 2. Thinking (designing a solution) 3. Experimenting with environment 4. Adding functionality 5. Parallelizing 6. Tuning 7. Debugging 8. Testing 9. Other	1. Adding functionality (serial code) 2. Parallelizing code 3. Improving performance (tuning) 4. Debugging: Compile-time error on previous compile 5. Debugging: Crashed on previous run (segmentation fault) 6. Debugging: Hung on previous run (deadlock, infinite loop, etc.) 7. Debugging: Incorrect behavior on previous run (logic error) 8. Restructuring/cleanup (no change in behavior or performance) 9. Other
2	1. Planning/Designing 2. Serial coding 3. Parallelizing the code 4. Testing the code 5. Tuning the code 6. Learning/Experimenting with environment 7. Other	1. Learning/Experimenting with compiler 2. Adding functionality (serial code) 3. Parallelizing code 4. Improving performance (tuning) 5. Compile-time error on previous compile 6. Run-time error on previous run (incorrect behavior, crashed, hung, etc…) 7. Other
3	1. Thinking 2. Serial coding 3. Parallelizing the code 4. Testing 5. Debugging 6. Tuning 7. Experimenting with environment 8. Other	1. Serial coding 2. Parallelizing the code 3. Testing 4. Debugging 5. Tuning 6. Experimenting with environment 7. Other
4	Unchanged	Unchanged
5	Unchanged	Unchanged
6	Unchanged	Compiler questions eliminated

Table 5.2 shows the evolution of the activity categories across the iterations. In the initial studies, the compiler and effort log activities were very different. The initial motivation for the differences was that the subject could specify development activities at a finer level of granularity at compile-time then when summarizing the data over an interval of time (For example, if a subject was debugging, at compile-time they would be able to specify the failure type: crash, hung, incorrect behavior).

We later concluded that even though there was the potential for collecting finer-grained data using compile-time questions than using the effort log, this finer grained data wasn't necessarily more useful, and it was actually more important to try and cross-reference the activities in the effort logs with the activities in the compile questions to see if they matched. The two activity lists finally converged in iteration 3. The only exception was the "Thinking" category, which assumes the subject is not working on a machine, and therefore does not exist in the list of compiler options.

As the research proceeded, it was not clear what criteria to use to identify the "right" set of categories. We were always concerned that asking the subjects a question at each compile might perturb their normal behavior. Since we could not sufficiently justify asking the questions, we dropped these questions in iteration 5 to avoid bothering the subjects. Instead, we decided to pursue a strategy where the activities are inferred automatically from the data that is collected automatically (e.g. from changes in the source code across successive compiles), rather than by explicitly asking the subject.

This research is still ongoing, and in iteration 7 (not covered in this work), other researchers have used an observational study method with multiple observers to try and evaluate workflow heuristics.

5.3.7 Performance measurement

While the focus of the research is largely on comparing effort, we must also measure performance in these studies since one of the main reasons to run on an HEC system is to improve performance.

Performance measurement is less straightforward than effort measurement because the performance of a particular program depends on many factors, such as inputs, machine architectures, compiler and library implementations. For the studies, we chose to use “speedup” as our measure of performance, sometimes referred to as “relative speedup” [Sahni96]. Roughly speaking, speedup is:

$$S = \frac{T_1}{T_N}$$

where T_1 is the execution time on one processor, and T_N is the execution time on N processors. Generally, the speedup should be less than N, although so-called “super-linear speedup” may occur due to memory/cache effects.

Throughout the studies, we have relied on self-reported performance data, since reporting performance is part of the assignments. However, we discovered in early studies that there was variation on how students reported effort. For example, we saw inconsistencies in the input data used and numbers of processors used for timing. In some cases, students reported their data only in graphical form, and it was difficult to read off the actual numerical values.

In later studies, we developed a template that made more explicit would performance data was being asked for to ensure more consistent reporting of performance data (see Appendix K).

In addition, "speedup" is a measure of scalability, rather than strictly a measure of performance. It is possible that, given two programs (A,B), program A exhibits better speedup on *N* processors than program B, but program *B* actually runs faster than program *A* [Sahni96].

Based on this, we initially decided, for purposes of analysis, to define "speedup" relative to the fastest serial implementation in the class, so it would capture performance rather than scalability (what Sahni calls "real speedup")

$$S = \frac{\min(T_1)}{T_N}$$

However, given that students often got relatively poor speedups even with the more forgiving notion of speedup versus single-processor performance, we decided to remain with relative speedup.

It is not strictly necessary to have the students report their performance to us, given that we collect all of the student submissions. However, we have not yet developed infrastructure for rerunning all of the codes. In addition, there is the problem of inconsistent program interfaces (e.g. different command-line arguments, data input formats expected). The template that we developed should alleviate this somewhat, as it makes explicit the desired interface, this does not always guarantee compliance.

In iteration 6, Viral Shah at UCSB developed a "harness," a set of skeleton code which all of the students must use as a basis for their program. This provides better enforcement to specified interfaces than simply a description in the assignment write-up.

5.3.8 Data storage

The classroom studies generated a large volume of data. The system for storing the data also evolved over the course of the study. All of the raw data (e.g. logs generated by wrappers, code submissions, student writeups) is simply archived as files on a server. However, this archive is not a convenient format for data analysis.

Initially, the raw data from the instrumentation was processed and stored in database tables to facilitate data analysis. This processing was done in a somewhat ad-hoc basis, and there was no proper database design. For example, separate tables were used for different classes, even if the type of data being stored was identical. In addition, the raw data was often processed with “one-off” scripts which were constantly being modified.

In iteration 5, a normalized system was developed to collect and store the data. Previously, the technical lead was responsible for coordinating all of the data involved in a study (e.g. background forms, reported effort forms, instrumentation logs). The new system was designed so that professors would be able to run studies with minimal assistance from the main research group, which was a goal of the later iterations in the methodology. This system, which we referred to as the Experiment

Manager[7] was designed and implemented in conjunction with visiting students from the University of Mannheim.

Using the Experiment Manager, a professor can specify the relevant details of a study, (e.g. assignment descriptions, due dates, programming models, machines). Students use the Experiment Manager directly to report their effort through the web interface (see Section 5.3.3), and they can upload their submissions when complete. Professors upload the data generated by the instrumentation directly to the Experiment Manager, where it is then stored in the back-end database. The data can then be examined by analysts.

5.3.9 Study setup

Initially, all of the study setup work was done by the author and Dr. Jeffrey Carver. This included an introductory presentation to describe the study to the subjects, distribution of forms, and installation of software on the target machines. For remote sites, the author traveled to the institution to present the study.

Once a professor had been involved in a study for at least one iteration, this level of participation in study setup was no longer necessary. The professor was able to give the introductory presentation (with the presentation materials that had originally been developed).

With the introduction of the Experiment Manager, there was no longer a need to distribute paper forms to the class. Instead, subjects were given a link to the

[7] http://care.cs.umd.edu:8080/umdexpt/cgi-bin/index.cgi

Experiment Manager (typically placed on the course home page, and also included in the introductory presentation), where they then created an account with the system.

The instrumentation package also evolved to a level of maturity where a third-party (e.g. the teaching assistant for a course) can install it on the target system without intervention by the author.

5.4 Observational studies

Three observational studies (Obs1, Obs-2, Obs-3) were conducted throughout the course of this dissertation where the author directly observed a programmer, in addition to the data collection tools mentioned previously. The purpose of these observational studies was to evaluate the accuracy of the algorithms used to estimate effort.

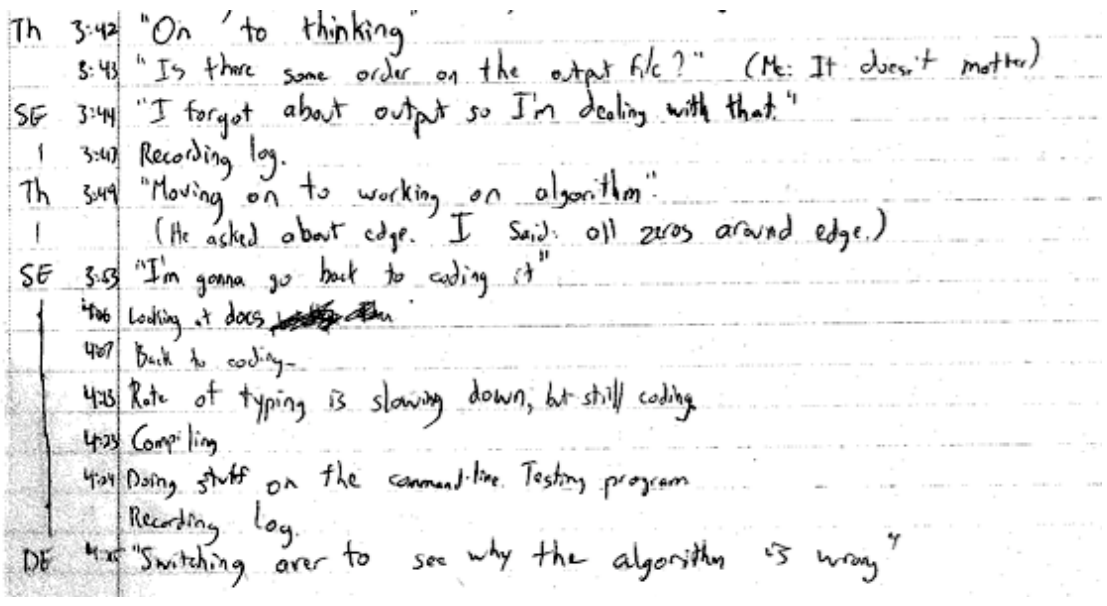
5.4.1 Protocol

In each of the studies, the author sat with an individual programmer and observed directly as the subject solved a parallel programming problem. The author kept notes about when the subject was working and what the subject was doing. The studies employed a talk-aloud protocol: the subject was encouraged to describe their development activities while they worked. If the subject's activity was not apparent, then the author would prompt the subject ("What are you doing now?")

5.4.2 Observer log

In the observer log, the author kept track of the different programmer activities, using a set of activity codes, as well as free-form text comments. In addition, the timing of the events was kept so that it could be cross-checked against the automatically collected data.

The format of the observer log changed between Obs-2 and Obs-3. In the first incarnation of the log, the author simply noted the start and stop times of the different activities in the margin, as shown in Figure 5.9.



Th	3:42	"On ' to thinking"
	3:43	"Is there some order on the output file?" (Me: It doesn't matter)
SE	3:44	"I forgot about output so I'm dealing with that"
|	3:47	Recording log.
Th	3:49	"Moving on to working on algorithm"
|		(He asked about edge. I said: all zeros around edge.)
SE	3:53	"I'm gonna go back to coding it"
	4:06	Looking at docs ~~[illegible]~~
	4:07	Back to coding
	4:23	Rate of typing is slowing down, but still coding
	4:23	Compiling
	4:24	Doing stuff on the command-line. Testing program
		Recording log.
DE	4:25	"Switching over to see why the algorithm is wrong"

Figure 5.9 Free-form observer log

The log was later modified to reduce the cognitive load on the observer. An interval coding log [Robson02] was adopted that broke up the day into five-minute time intervals. The observer coded each interval, as show in Figure 5.10.

Study
Date:

	Un	Fs	Fp	Vv	De	Os	Op	Ex	Ot	Notes
9:00 AM										
9:05 AM										
9:10 AM										
9:15 AM										
9:20 AM										

Figure 5.10 Structured observer log

As shown in Table 5.3, the set of activities used in the log varied across the observational studies. The activities in Obs-1 and Obs-2 match the activities used in the classroom studies. In Obs-3, a new set of activities was developed in conjunction with the subject of the study, based on the subject's opinion of the different development activity.

Table 5.3 Activities in observational studies

Study	Activity	Code
Obs-1	Thinking (understanding the problem)	UN
	Thinking (designing a solution)	DE
	Experimenting with environment	EX
	Adding functionality	FU
	Parallelizing	PA
	Tuning	TU
	Debugging	DE
	Testing	TE
	Other	OT
Obs-2	Thinking	TH
	Serial coding	SE
	Parallelizing	PA
	Testing	TE
	Debugging	DE
	Tuning	TU
	Experimenting with environment	EX
	Other	OT
Obs-3	Understanding the problem	Un
	Adding functionality – serial	Fs
	Adding functionality – parallel	Fp
	Testing/V&V	Vv
	Defects	De
	Optimizing – serial & parallel	Os
	Optimizing – parallel only	Op
	Experimenting with environment	Ex
	Other	Ot

The automatic instrumentation employed was the same that used in the classroom studies (see Section 5.3.4). The observer's effort log was treated as a "gold standard" of the measure of effort, which was then used to evaluate the effort as estimated by the automatically collected data. The coding and notes in the log were used to identify the development activities where the observed effort did not match the estimated effort.

The environment in which the studies took place changed from one study to the next, in an effort to increase the external validity of the study. Obs-1, which was a pilot study, took place in a room we had reserved for the study (an office of one of the researchers). Therefore, the subject was in an unfamiliar environment during the development process. This study was concluded in a single session, with the subject working continuously throughout that session.

In Obs-2, the study took place in the subject's office, rather than in a separate room. The study took place in multiple sessions across several days. The author would schedule sessions with the subject, and then come to the subject's office to observe. The subject worked in continuous sessions of about 2 hours per session.

In Obs-3, the study was conducted so that the subject's work habits most closely resembled their habits if the study was not being conducted. The subject's work station was conveniently located across from the author's work station, so there was no need to schedule appointments. In addition, the subject installed VNC[8] software on his machine so that the author could observe the subject's screen without looking directly over the subject's shoulder. With this organization, the subject was potentially much less aware of being observed (and, in fact, would sometimes forget

[8] http://www.tightvnc.com

about being observed, as evidenced by his occasionally checking personal email during the study).

5.5 Folklore gathering

We use the term “folklore” to refer to beliefs held by individual practitioners or communities of practitioners. One of the research efforts was to attempt to elicit this folklore and record it explicitly. We believe that make this type of tacit knowledge explicit is an important starting point when conducting empirical software engineering research in a new domain, as it has the potential to provide a rich set of hypotheses that will help guide the researcher in designing studies.

5.5.1 Initial seeding through background research

We began by building an initial set of folklore elements to serve as the basis of discussion for future interactions with practitioners, in iteration 3. Dr. Jeffrey Hollingsworth, our domain expert, used his lecture notes to build an initial list of folklore elements.

5.5.2 Focus group

The initial list was used as a starting point for a telecon-based focus group. The participants in the focus group were members of the HPCS Development Time Working Group. These were predominantly HEC professors who were participating in the classroom studies, but it also included other practitioners. The list was sent out over email in advance of one of the monthly telecoms. During the telecon, we went over each element of folklore that had been collected, and the participants discussed the conditions for which each element of folklore was true, and some new elements of folklore were created.

5.5.3 Surveys

Surveys were distributed to practitioners at one of the semi-annual DARPA HPCS meetings (see Appendix E). The surveys contained the elements of folklore that had been collected and asked the participants whether they agreed or disagreed with the element as stated, and to explain their answer. In addition, it asked for some information about their experience. In an attempt to correct for bias due to the wording of the questions, we created two surveys. For each element of folklore, we tried to generate a logical inverse. Each survey contained half of the original folklore elements, and half of the logical inverses.

5.5.4 Interviews

The final method we employed for folklore gathering was direct interviews with practitioners (The interview guide can be found in Appendix G). These were semi-structured interviews that involved for individuals: two interviewers and two interviewees. Two interviewers were used so that one could serve as a scribe, and could ask additional questions if the primary interviewer forgot to ask a question. We interviewed practitioners two at a time to foster interaction among them and draw out more information than if they were interviewed individually.

5.6 ASC-Alliance case studies

The ASC-Alliance case studies were studies of large-scale, mature (greater than 5 year) HEC projects. Before the ASC-Alliance case studies were done, we had no information about the nature of the software development process for HEC projects in an academic setting. Three of these studies were conducted in iteration 6, and two were conducted subsequently.

5.6.1 Initial survey

We began by sending a simple two-page questionnaire to each center (see Appendix H) to get some sense of the scope of the project. Once this was returned, we sent the interview guide to each project in advance, so that they would have an opportunity to reflect upon the questions.(see Appendix I).

5.6.2 Telephone interview

After the survey was returned to us, we arranged a telephone interview with one member of the project who was very familiar with the software itself. At least two interviewers were used each time: one interviewer asked the questions from the guide and the other interviewer kept notes. The second interviewer would occasionally ask clarifying or follow-up questions in case the first interviewer missed a crucial piece of information. In addition, interviews were tape recorded (with the consent of the interviewees). The interviews are rarely transcribed entirely, because of the time required to do so. Instead, the tape recordings were used to augment the notes in areas where there were gaps or certain issues were unclear.

5.6.3 Summary

After the interview was completed, we produced a summary of the content of the interview based on the notes. This summary was then sent back to the original interviewees to ensure that all of the information had been captured adequately. Usually, several minor corrections were made by the interviewees.

5.6.4 Synthesis

Finally, these summaries were combined into a single paper that represented content from each of the centers. This paper was then passed around to each project for approval.

5.7 Classroom case studies

Case studies were also performed on class projects in one of the classroom studies in iteration 6 (B-F05). The purpose of conducting the classroom case studies was to understand where students spent their time on larger projects, to characterize the differences between class assignments and larger projects, and to determine whether novice parallel programs could achieve their performance goals on small HEC projects.

The primary method for studying the classroom studies was through semi-structured interviews, although we also collected the final reports, final source code, and instrumented data when possible (no self-reported effort logs were used).

The interviews were conducted with all of the project members at the same time. There were two interviewers (primary interviewer and scribe). The interview guide can be found in Appendix J.

5.8 Productivity modeling

The goal of productivity modeling is to develop metrics for HEC systems (including both hardware and software aspects) that capture the notion of “programmer productivity”. This is in contrast to traditional metrics which focus exclusively on issues related to machine performance on a set of benchmarks such as the NAS Parallel benchmarks [NAS-95-020]. The reason for the productivity modeling is to help procurement agents identify the machines that will maximize the productivity of the programmers who will be using the machine.

As of this writing, the modeling has not reached the level of maturity where studies are being designed specifically to validate the model. However, data from the other efforts has been used to verify that the model is feasible in the sense that metrics can be generated from the collected data. Zelkowitz et al. proposed a model and used the effort and performance data from the classroom studies to validate that the model could be applied to real data [Zelkowitz05].

Chapter 6 Results

6.1 Introduction

In this chapter, we select some results from the research to show how applying the methodology builds knowledge iteratively over time.

6.2 Building confidence: programming model and effort

One of our initial goals was to measure the effect of parallel programming model on programmer effort. By using an iterative approach, we were able to build up confidence in the results by improving the quality of individual studies as we gained maturity. A note on reporting results: following Cohen's recommendations, we generally favor 95% confidence intervals (abbreviated CI), rather than p-values [Cohen94]. Confidence intervals show the size of the effect being measured and the uncertainty in the estimate, along with providing the same information as p-values. For a 95% confidence interval, there is a 95% *a priori* probability that the confidence interval will include the true value of the outcome variable of interest. (typically, the difference between means of two populations). If this interval does not include 0, this is equivalent to a p-value of less than .05.

Brief descriptions of each assignment used in the classroom studies can be found in Appendix B.

6.2.1 Initial attempt

In study A-F03 of the first iteration, A paired t-test showed that statistically significantly less effort was required for the OpenMP/SWIM problem compared to the MPI/game of life problem (95% CI: 1.9-12.7 hours). Our confidence in the validity of these results was low because the nature of the problems was so different.

6.2.2 Better comparisons and looking across classes

In the second and third iterations, compared across models within the same problem. These were within-subject comparisons, where the students solved the same problem in multiple models. We also compared the same problem and model across classes

Measuring effort differences within subjects

Here we begin to see how the difference in MPI/OpenMP effort changes when the students are solving different types of problems. Table 6.1 shows the results of comparing MPI and OpenMP effort across the different problems. The results tell a different story than in the previous iteration. The game of life was easier in OpenMP than MPI, but in one of the two Buffon-Laplace assignments, MPI required less effort, and the difference was not statistically significant in the other Buffon-Laplace problem or in the grid of resistors problem.

Confidence is still low because there was no control for ordering in these comparisons. For example, the reduction in effort may be because the students solved the problem with MPI before solving it in OpenMP, and this additional experience with the problem reduced the OpenMP effort.

Table 6.1 Iteration 2: Comparing MPI/OpenMP effort across problems

Problem	Statistically significant?	Effort savings vs. MPI
Game of life (E-S04)	**yes** , paired t-test CI=[1.2,8.5] hours	49%
Buffon-Laplace (D-S04)	**no** , paired t-test CI=[-1.8,3.1] hours	24%
Buffon-Laplace (E-S04)	**yes,** paired t-test CI=[-2.1, -0.1] hours	-79%
Grid of resistors (D-S04)	**no** , paired t-test CI=[-1.4,4.0] hours	54%

Comparing across classes

We would like to be able to run studies by comparing across classes. However, in such a quasi-experiment, we want some confidence that any differences observed in the dependent variables are not due to differences in the two student population (e.g. quality of students, quality of instructor, development environment).

Table 6.2 Iteration 2: Comparison across classes

Problem	Studies	Statistically significant?
Game of life / MPI	A-F03,B-S04,E-S04	**no** (ANOVA, p=0.40)
Buffon-Laplace / MPI	D-S04, E-S04	**no** (t-test, CI=[-2.5, 5.1] hours)
Buffon-Laplace / OpenMP	D-S04, E-S04	**no** (t-test, CI= [-2.7,1.8] hours)

The results shown in Table 6.2 provide us with some confidence that the differences across classes are small, as we did not obtain statistically significant results in any of the three problems solved across classes.

6.2.3 Beginnings of random assignments, comparing across classes

In the fourth iteration, we began to randomly assign students to treatment groups. In D-S05, the students had to use both MPI and OpenMP to solve the problem, and were told which one to do first. In E-S05, the students had to use two models for each assignment, one which was assigned (either MPI or UPC/CAF), and the other one they could choose. Table 6.3 shows the result of these studies. Unfortunately, the introduction of controls reduced the power of the study by reducing the sample size. Since students could choose one of the two models, there were fewer data points for comparing across models.

Table 6.3 Iteration 4: Comparing MPI/OpenMP effort across problems

Problem	**Studies**	**Statistically significant?**	**Effort saved vs MPI**
Parallel matrix power	D-S05	**no**, paired t-test, **CI**=[-1.0,9.9] hours)	54%
Game of life	E-S05	**no**, paired t-test **CI**=[-1.5,38.7] hours)	88%
Sparse conjugate gradient	E-S05	**no**, paired t-test, **CI**= [-16.2,38.5] hours)	61%

We were also able to do a comparison of XMT-C and MPI by comparing across classes, using the sparse-matrix multiply problem from D-S04 and C-S05. The results were statistically significant, (CI: 2.4- 8.2 h), with a mean effort savings of 50%.

6.2.4 Full control for ordering

By the sixth iteration, we had a fully controlled design where half of the students solved the problem in MPI and half solved in OpenMP, and then they submitted the assignment, and switched, and submitted the assignment again. Therefore, we had high confidence that there was no ordering effect, and that the protocol was followed by the subjects.

We also witnessed, for the first time, a statistically significant difference across classes of the same type of problem (sharks & fishes, MPI, G-F04, B-F05). The assignment descriptions were not identical: professor B had modified the assignment description from G-F04 to make the problem deterministic.

6.2.5 Summary

Figure 6.1 and the associated Table 6.4 show the results for studies that directly compare MPI and OpenMP. The figure shows confidence intervals for each comparison, as well as the percentage of mean effort saved by using OpenMP instead of MPI. The confidence intervals are ordered by the difficulty of the problem (as estimated by mean MPI effort). The table describes which programs and studies were involved, whether the comparisons were within-subject or between-subjects and, in the case of within-subject studies, if there were controls for the order in which the subjects solved the two problems.

Based on these results, it is difficult to judge how much impact the type of problem had on the amount of effort saved. The embarrassingly parallel Buffon-Laplace problem (labeled "1" in the figure) stands out as a case where MPI actually required less effort than OpenMP. This was most likely due to issues related to the random number library, which resulted in unusually poor OpenMP performance.

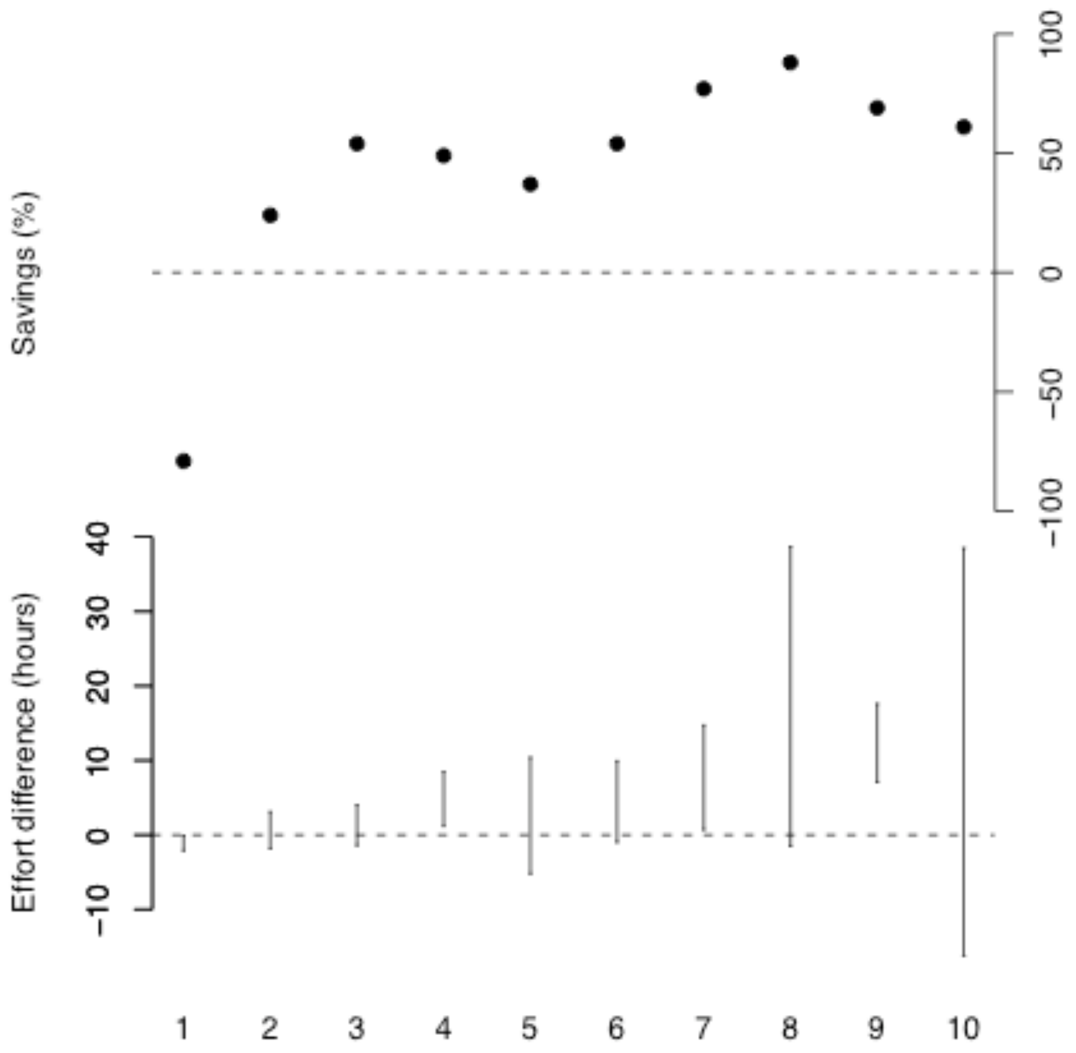


Figure 6.1 MPI-OpenMP comparisons

Table 6.4 Summary of MPI-OpenMP comparisons

ID	Problem	Study	Comparison	95% CI	% Savings	Control for ordering?
1	Buffon	E-S04	Within	[-2.1,-0.1] h	-79%	No
2	Buffon	D-S04	Within	[-1.8,3.1] h	24%	No
3	Resistors	D-S04	Within	[-1.4,4.0] h	54%	No
4	Life	E-S04	Within	[1.2,8.5] h	49%	No
5	Sharks-fishes	G-F04	Within	[-5.2,10.4] h	37%	No
6	Matrix-power	D-S05	Within	[-0.98,9.9] h	54%	Yes
7	Matrix multiply	G-F04	Within	[0.7,14.7] h	77%	No
8	Life	E-S05	Between	[-1.5,38.7] h	88%	N/A
9	Sharks-fishes	B-F05	Within	[7.1,17.6] h	69%	Yes
10	Sparse CG	E-S05	Within	[-16.2,38.5] h	61%	No

Comparisons between other models (MPI-UPC/CAF, UPC/CAF-OpenMP, MPI-XMT-C) are shown in Table 6.5.

Table 6.5 Other comparisons

Models	Problem	Study	Comparison	95% confidence intervals	% savings	Control for ordering?
MPI: UPC/CAF	Life	E-S05	Between	[-8.4,19.3] h	43%	N/A
MPI: UPC/CAF	Sparse CG	E-S05	Within	[-0.6,29.4] h	41%	No
UPC/CAF: OpenMP	Life	E-S05	Within	[-5.9,16.6] h	59%	No
MPI:XMT	Sparse matvec	D-S04, C-S05	Between	[2.4,8.2] h	50%	N/A

We also saw evidence that suggests that comparisons across classes are valid, provided the assignments are as similar as possible. If an assignment is modified, it may result in a very different effort profile, as we saw for the sharks and fishes problem. This result also suggests that a simple characterization of the problem based on communication pattern (e.g. “nearest neighbor) may not be sufficient for predicting the impact of a programming model on the effort, since changes which do not affect the classification can still have measurable effects on the effort.

6.3 Novices and the role of experience

When running human-subject studies, software researchers must be concerned with how subject experience affects the results of a study [Wohlin04, Host05], especially when using students as subjects. Pancake and Cook have suggested that the typical novice/expert classification may not apply to HEC practitioners because practitioners may have years of sequential programming experience but may program only infrequently. Therefore, they are neither “novices” nor “experts” [Pancake94].

6.3.1 Background questionnaire

Our initial attempt, beginning with the first studies, was to use a background questionnaire to measure programmer experience. Measuring the effect of experience through statistical analysis proved to be too difficult because of the multi-dimensional nature of experience. Among experience variables we tried to capture were:

- Experience with the particular programming model (e.g. MPI)
- Experience with concurrent programming

- Experience with the base language (e.g. C, Fortran)
- General software engineering experience
- Experience solving the particular problem
- Experience with numerical algorithms in general
- Academic background (field of study)

With our small samples sizes, it is was not possible to do a proper analysis of variance to identify the effect of experience variables.

6.3.2 Comparing with experts

Since we could not measure the effect of experience within our subject pool, we sought programmers with greater experience that we could use to compare to the students. Our original plan was to have the professors solve the assignments, although all proved reluctant to do so, and claimed that their graduate students would be better candidates. In two studies, H-F04 (LU decomposition with OpenMP) and C-S05 (sparse matrix-vector multiply with XMT-C), the professors volunteered a graduate student to implement the problem. Unfortunately, we were only able to obtain effort in the case of H-F04. In that case, the effort was consistent with that of the students. In both cases, the performance of the "expert" code was greater than that of the fastest student code. In the case of H-F04, this difference was on the order of 10x, whereas for the case of C-S05, the difference was only on the order of about 10% faster than the fastest student code. It is worth noting that the H-F04 problem was specifically a tuning problem, and the expert's research area was compiler optimizations. We hypothesize that expertise has a much greater effect on performance than on effort, although our confidence is low because we had only two examples of expert performance and one of expert effort.

6.3.3 Experience with a particular problem

By iteration 6, we were able to exert sufficient control in a study (B-F05) that we could measure the effect of one aspect of experience: having solved a problem before in a different parallel programming. Figure 6.2 is an interaction diagram which shows the mean programming effort for four groups: MPI/first-time, MPI/second-time, OpenMP/first-time, OpenMP/second-time. The data shows that for both MPI and OpenMP, having solved the problem before in a different model reduces the mean effort. However, the mean effort for solving MPI the second time is larger than the mean effort for solving OpenMP the first time, suggesting that programming model has a larger impact than problem experience. In addition, there is no interaction between the variables: the difference in effort between MPI and OpenMP is the same, provided both groups are either solving the problem for the first time or the second time.

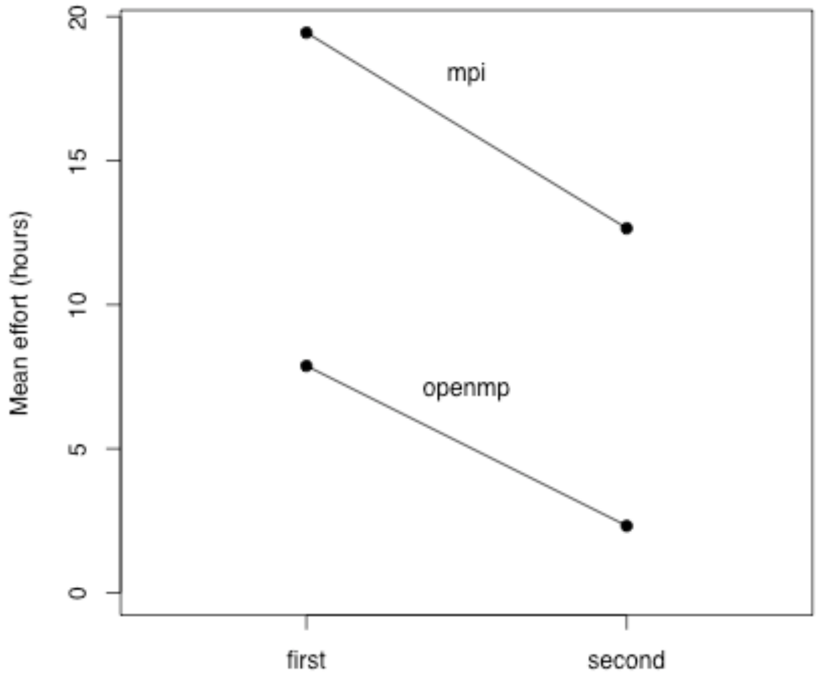


Figure 6.2 Effect of experience on effort

Figure 6.3 shows the equivalent interaction diagram for speedup. Here a very different picture emerges: for those who solved the problem with MPI first, their OpenMP programs went faster. However, solving the problem with OpenMP first had no effect on MPI performance. We see a surprising interaction: for people solving the problem the first time, MPI is faster, and for solving the problem the second time, OpenMP is faster.

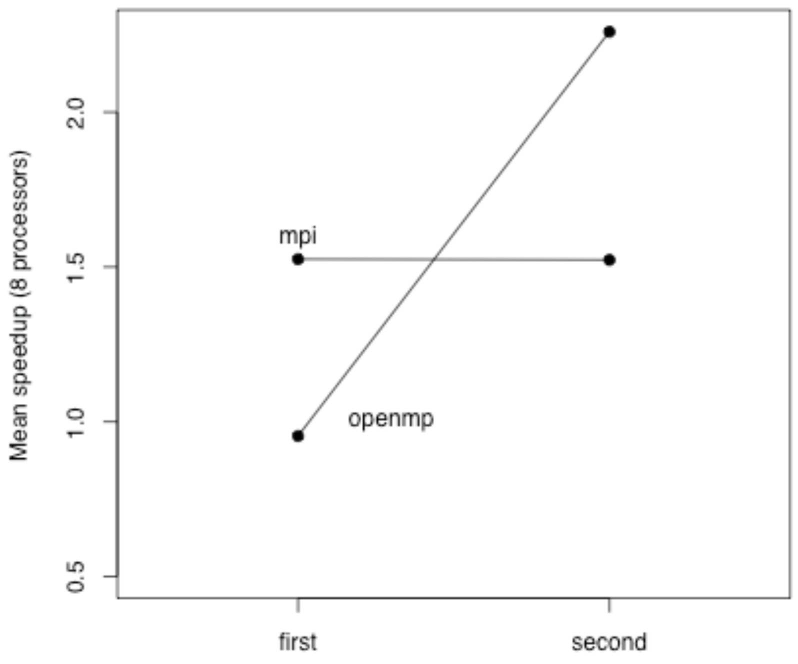


Figure 6.3 Effect of experience on speedup

6.3.4 Experience needed for large projects

In the ASC-Alliance case studies conducted in iteration 6, we did not specifically address issues related to experience. However, on all projects we saw evidence that graduate students were involved in development, although the project leads were never graduate students and always had many years of experience. One interviewee stated that

"For the professor whose job it is to turn out students, the correct [productivity] metric is the length of time from when a grad student finishes the second year of coursework to [when he becomes] 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."

6.3.5 Summary

To summarize the results about experience: we saw that we were not able to measure the effect of experience across students by using a background questionnaire. The professors did not view themselves as "expert" programmers but regarded their own grad students as such, and believed that the best grad students in their class were comparable to experts. We saw some weak evidence that suggested that experts would achieve better performance than novices, but not necessarily with less effort. For a particular problem, we saw a measurable reduction in effort with experience with having solved that problem before in a different model, but the effect on performance varied depending upon the previous model. Finally, from the ASC-Alliance interviews we saw that graduate students can become productive contributors to HEC projects, but this takes a certain amount of time (presumably, months to years of training).

6.4 Focusing in: Defects and debugging

Our initial collection of folklore provided us with two seemingly contradictory elements:

F.1 It's easier to get something working using a shared memory model than message passing.

F.5 Debugging race conditions in shared memory programs is harder than debugging race conditions in message passing programs.

These contradictory elements were captured by an interview with Jeff Vetter from ORNL in iteration 6, who described OpenMP as: "easy to parallelize, difficult to debug".

From the initial iteration, we tried to capture the time spent in various activities (including debugging). However, because of the complexities involved in trying to capture workflow (see Section 5.3.6), we were not able to get an accurate sense of how much time people spent debugging.

From the folklore gathering interviews in iteration 5 we received some contradictory information in this area. The following question was asked in the folklore interviews: "*Which model [shared memory or distributed memory] makes race conditions easier to debug and under what conditions*". Jeff Vetter stated that race conditions were more difficult to debug in OpenMP, and John Feo made the stronger statement that *no* race conditions occur in message-passing. In contrast, John Gilbert claimed that anything in shared memory is easier to debug. Declan Murphy gave the most nuanced answer:

"*Race conditions are very different in both models but easier to deal with in shared memory programming. I think a lot of people think the opposite and I think the reason they believe this is just because when you do message-passing you spend so much processing time in the message-passing code and so little time in the actual application-level processing of the message that it is harder to hit race conditions in message-passing, it doesn't mean they don't exist.*

In message-passing, race conditions are rare but when they happen they are very *difficult to debug.*

It is easier to get something working in message-passing but if you hit a bug it is hard to solve. Looking at shared-memory code, potential race conditions are pretty obvious but message-passing code is much harder to understand."

To study debugging in more detail, we began to collect defect data in the classroom studies. In iteration 3, we piloted a defect log in H-F04. In iteration 4, we used this log to compare MPI and OpenMP defects, and did not find a statistically significant difference in time-to-fix between MPI and OpenMP defects.

Because the previous results and folklore were so ambiguous, Taiga Nakamura began to study defects in more detail, and developed a methodology for classifying and identifying defects using the data that we had collected from previous studies, and generated a set of common defect categories [Nakamura06]. In iteration 7[9], we tried to apply the knowledge developed by Nakamura by introducing it to the students in C-S06 and trying to measure the effect of this treatment against historical data. Due to circumstances outside our control, we were not able to conduct this study and it will be attempted again in the next iteration.

Three of the classroom case studies conducted in iteration 6 used OpenMP. Two of these projects discussed race conditions that had been encountered in their OpenMP code. For one of the projects (object tracking), they identified which

[9] This iteration was ongoing as of the writing of this dissertation and is therefore not covered in detail.

variables should be private and which should be shared through trial and error, which suggested that they had difficulty with debugging race conditions.

In the ASC-Alliance case studies conducted in iteration 6, we saw a very different view of debugging. In one of the interviews, the project members were more concerned with "algorithmic defects", trying to determine if a particular algorithm had the right qualities to be used in the application (e.g. numerical stability). They saw this as a much more difficult problem than errors in implementing the algorithm, which they referred to as "software defects". In the two ASC-Alliance case studies conducted in iteration 7, we asked about this phenomenon and got similar responses. In addition, we asked the two projects whether they used an environment such as MATLAB for prototyping new algorithms. Members of one project responded: "*Of course we do*", and the other responded: "*No, [we] don't. For the most part, [we] do it in Fortran, because historically [we've always] done it in Fortran.*"

6.5 Understanding the role of "tuning"

From the initial stages of the research, we tried to capture the time spent in different development activities. One of the activities that were of interest was what we called "tuning", time spent improving performance on a given machine. Sahni and Thanvantri state that *"To change parallel computers usually requires that users rewrite or at least* ***retune*** *all programs according to such features as memory organization, interconnect topology and technology, ratio of computational speed and communication bandwidth, number of available processors, and the ability of the processors to overlap computation and communication"* [Sahni96] (emphasis added). This suggests that there is an initial tuning phase during code development, and an additional tuning phase during porting.

A survey of practitioners conducted by Pancake and Cook revealed that programmers spend 12-18% of their parallel programming effort on "tuning program performance" [Pancake94].

6.5.1 Folklore and classroom studies

Folklore collected in the initial iteration suggested that tuning is an important development activity, although we knew little about how it was done or how much time it took:

F.2 Once you learn it, identifying parallelism is easy, but achieving performance is hard. For example, identifying parallel tasks in a computation tends to be a lot easier than getting the data decomposition and load balancing right for efficiency and scalability.

F.3 Usually, the first parallel implementation of a code is slower than its serial counterpart.

In A-F03 of iteration 1, we saw no statistically significant correlation between performance and effort across the students in the class (p=0.425). This suggested that, whatever it was the students were spending their time doing, it wasn't (successfully) getting their code to run faster. In later iterations, we saw several assignments where less than half of the students got any speedup at all. We hypothesized that that the students were not able to get good performance because they were either unwilling or unable to tune their correct but slow parallel programs.

In iteration 2, the students were given a tuning-specific assignment (F-S04: LU decomposition), where they were given an existing serial program and asked to make it run as fast as possible on a particular architecture. All of the students were able to achieve speedup. It is notable that this was a compiler course rather than an HEC course, and the students were taught about specific compiler optimizations that they performed by hand. This suggested that students were capable of tuning for better performance if instructed appropriately.

We explored tuning issues in the folklore interviews in iteration 5. Bryan Biegal of NASA observed that achieving performance gain is more difficult on small codes, and correctly parallelizing is more difficult on larger codes. He described his personal workflow as developing a prototype in MATLAB, using a staff programmer to develop a baseline parallel version, and then using an expert to tune the parallel code. John Gilbert of UCSB described how he had witnessed many projects where the development cycle is: identify parallelism, code, tune, give up, and start again with a different algorithm. He also mentioned that when starting with a sequential code, the first thing that is typically done is gaining about 5x speedup in the sequential code before improving performance through parallelism.

6.5.2 Classroom case studies

In the classroom case studies conducted in iteration 6, we asked the students to tell us where they spent their time in terms of planning, serial coding, parallel coding, and tuning/optimizing. While several groups mentioned some non-zero value for tuning, when pressed further we discovered that they did not tune their code, at least not in the way that we understood the term. For example, in B-F05-C1 (chess), the students estimated they spent about 35% of their time tuning. When we asked them specifically about this tuning activity, they described how they tried to implement pruning in their search algorithm. This is a development activity designed to improve performance, but it is very different from the type of activity implied by [Sahni96]. In B-F05-C2, when initially asked about effort breakdown, the students estimated that 30-35% of their time was spent in parallelizing and tuning, with about a week parallelizing and 4-5 days tuning. Yet, they responded "yes" when asked if they got performance improvement on their first correct parallel solution, and later said they did not try and tune after the code was working. (They did, however, use profiling tools in the sequential MATLAB code that they began with).

In B-F05-C4 (porting numerical algorithms from threads to MPI and OpenMP), one student estimated 10% of his effort was spent on tuning, but when asked about details, admitted that no actual tuning was done: the OpenMP implementations simply worked without being tuned, and the MPI implementations had very poor performance. For one problem (radix sort), to improve performance they simply re-implemented the entire algorithm from scratch rather than attempt to tune the existing algorithm.

The other two projects (B-F05-C-3, B-F05-C5) estimated roughly 5% of their time was spent tuning, and when asked for details, suggested that no actual tuning was done because of time constraints. However, both projects were happy with the performance of their resulting code.

6.5.3 ASC-Alliance studies

In the ASC-Alliance interviews, the interviewees interpreted “tuning” as machine-specific tuning. None of the projects we spoke to made modifications to “retune” to a particular architecture as implied by [Sahni 96], because they did not feel that it was worth the effort to tune for particular platforms. Their codes as implemented were suitably portable that significant modifications to the code were not necessary when porting to a new platform. Porting effort consists largely of getting the code to build properly on the machine. The following quote from one of the project members summarizes the tuning sentiment across all projects:

“*The amount of time it takes to tune to a particular architecture to get the last bit of juice is considerably higher than the time it takes to develop a new algorithm that improves performance across all platforms.*”

When asked “What is more difficult, achieving parallelism or performance?”, some project members were confused by the question. Upon further discussion, we discovered that they did not make a distinction between the two. They did not write a correct parallel code and then improve the performance through tuning. Rather, they knew the code well enough that they were able to achieve good performance on the first parallel implementation.

6.5.4 SDSC Consulting study

We are participating in an ongoing study at SDSC where consultants from the Strategic Applications Collaborations (SAC) program work with an existing high priority code project to help it run on one of the large machines. Initially, we believed that tuning to a particular architecture would be a central part of this work. However, the optimizations employed by the consultant were largely low-level changes that should improve performance across all platforms, such as inlining, taking constants out of loops, and reducing memory usage. This is consistent with John Gilbert's observation about the role of sequential tuning in the folklore interview. The only parallel optimization mentioned was a change in the initialization algorithm from point-to-point to block communication, and this does not appear to be a machine-specific optimization, but should improve performance across all machines (assuming a reasonable library implementation of the block communication functions).

6.5.5 Summary

To summarize, we initially believed that tuning played an important role in the development of HEC codes, and that the tuning activity was often a machine-specific activity. From the classroom studies, we believed that that students were not tuning their code (except in one tuning-specific problem), which explained in their poor performance, in contrast with tuning in "real" projects. The folklore that we initially collected reinforced this belief in the role of tuning in larger projects.

However, as we looked at larger projects, we saw less and less evidence for tuning in general, and machine-specific tuning in particular. In the student case study projects, unlike in many of the class assignments, the students were able to achieve reasonable performance, despite an absence of tuning. When looking at the ASC-Alliance projects, we saw an explicit aversion to machine-specific tuning. Instead, we saw an emphasis on more efficient high-level algorithms, and evidence that the programmers were able to write efficient parallel code without the need for a specific "tuning" phase. Even when we examined one of the SDSC consultants working on a specific "tuning" project, we saw very little evidence of machine-specific tuning. Instead saw low-level optimizations that were mostly serial optimizations and not machine-specific.

These results suggest that the term "tuning" should be more clearly defined when pursuing further research, especially given that students in the classroom projects initially claimed some "tuning" effort when no such work was actually done, and given the much higher tuning estimates (12-18%) that were observed in the survey in [Pancake94].

These results also suggest that the only people doing machine-specific tuning may be either people who are given code written by others (as mentioned by Bryan Biegal), and the vendors. This is reinforced by a comment made by John McCalpin of IBM[10] about the scalability of OpenMP (emphasis added):

*"In my previous job at SGI, my team regularly showed effective scaling to 128 processors with OpenMP codes, and many of these codes would clearly have scaled farther if larger systems had been easily available… [However], **effective scaling with OpenMP required an unusually comprehensive understanding of all of the components of the system**: the algorithm, the implementation, the compiler, the run-time libraries, the operating system, and the hardware. While my performance team was regularly capable of pulling all these items together for a successful*

[10] Sent to HPCS Development Time mailing list, 4/24/05

demonstration, ***we found that very few customers were able to reproduce our success.***"

Our results are consistent with this observation, given that end user programmers do not wish to invest effort into understanding components such as the compiler, run-time libraries, operating system, and hardware.

Chapter 7 Organizing the results into an experience base

In this section, we discuss the methodology used to organize the results of the studies into an experience base. Given the large number and variety of studies involved in this research, the process of organizing the results is itself a significant task.

7.1 An embarrassment of riches

One assertion of this dissertation is that building knowledge about a software domain requires a "broad" approach. While such research may involve controlled experiments, much of it will also involve more exploratory, qualitative strategies. Since building knowledge in a new domain involves conducting many different studies, we found that simply managing results of these studies became a significant problem in its own right. In particular, these studies generate a large number of hypotheses. We found it difficult to simply keep track of all of the hypotheses and their origins. In addition, we sometimes found it tempting to pursue hypotheses which seemed interesting, but which did not have any relevance to practitioners. Therefore, we saw a need for some mechanism that forces the researcher to consider hypotheses in terms of their consequences to stakeholders in the domain.

The idea of an "experience base", where the results of studies are stored in a repository (as opposed to simply a collection of publications) is not a new one. Basili et al. define an experience as "*a repository of integrated information, relating similar projects, products, characteristics, phenomena, etc.*" [Basili94]. Our vision of an experience base in this context is more modest. Here, we focus on the traceability of hypotheses rather than as a comprehensive repository of artifacts and models, which would be beyond the scope of this work.

7.2 Proposed organizational scheme

To deal with the problems described above, we propose a methodology for building an experience base to capture the evolution of hypotheses. Using this methodology, researchers build a “chain” of evidence where hypotheses are linked together through empirical studies. This methodology supports an iterative approach, so that the results of future studies can be incrementally added to the experience base. In addition, it requires that the researcher make explicit both the sources of a hypothesis and its implications, so that this information is preserved. Our scheme consists of three main entities:

- hypotheses
- evidence
- implications

Hypotheses are modified and created through *evidence* that is generated from studies. *Implications* describe the consequences of the hypotheses for practitioners. In the following sections, we will describe specifically what information is associated with these entities and provide examples from the studies described in the previous chapter.

7.3 Process of adding to the experience base

There are two typical use cases for a researcher to add information to the experience base

1. After a study has been completed and the results have been analyzed
2. When designing a study

In this section, we provide a high-level overview of these cases. Section 7.5 provides more detailed examples from the studies conducted as part of the dissertation.

7.3.1 After a study

The results of a study are added to the experience base as "evidence" (see the next section for details). Some outcomes from a study may result in the modification of existing hypotheses, or may result in the generation of new hypotheses. If an existing hypothesis is modified, the evidence in the base that is associated with that hypothesis is also modified to incorporate the new results. If a new hypothesis is created, then the results of the study are associated with the new hypothesis to describe its origin and the implications of the new hypothesis must be specified. If a hypothesis is modified, the implications associated with this hypothesis should be updated.

7.3.2 Before a study

One benefit of the experience base is that it helps identify hypotheses for future study. Before a study is conducted, there is no new evidence to add to the experience base. However, the researcher should have some initial hypotheses in mind before beginning the study. Even if the study is exploratory, the assumptions of the researcher should be documented as hypotheses in the base. This will avoid the problem of “obvious” results that were not actually believed when the study began.

7.4 Details of the scheme

In this section, we describe the entities of the scheme, as well as the relationship among entities. The detail in this section should be sufficient for a developer who wishes to build a software system to support this methodology.

7.4.1 Hypotheses

A hypothesis is used in the usual sense: an assertion about some aspect of the domain. For example: Shared memory programming requires less effort than message-passing programming. Table 7.1 shows the structure of the hypothesis entity.

Table 7.1 Hypothesis

Field	Description
Label	Short identifier
Description	Description of the hypothesis
Classification	Type of hypothesis
Topics	Set of topics that this hypothesis is related to
Implications	Set of implications implied by this hypothesis
Evidences	Set of evidences that support this hypothesis

In our scheme, hypotheses are classified according to how they were generated, as shown in Table 7.2. Hypotheses are generated through evidence from empirical studies, except for the initial set of hypotheses which are based on the researchers’ preconceived notions.

Table 7.2 Classification of hypotheses

Classification	Description
Initial	Did not originate in any particular study.
Refinement	Refinement of an existing hypothesis. Typically, more detail is added and the context is restricted.
Explanatory	Possible, unverified explanation for an experimental result.
Generalization	Generalization of a previous hypothesis, increasing its scope.
Observation	New hypothesis based on an observation made in a study that was not specifically being tested for in the study. This would include "folklore" collected from interacting with practitioners.
Alternative	A hypothesis that invalidates a previous hypothesis due to new evidence.

There are no restrictions on the nature of a hypothesis. It may be very narrowly defined, so that it could be directly testable in a controlled experiment, or it may be much broader but vague, such as an element of folklore.

7.4.2 Evidence

The role of evidence in this scheme is to describe the source and support of each hypothesis. Evidence is always associated with empirical data. This data my come from a formal source such as an empirical study, or an informal source such as a casual discussion with a practitioner. The purpose of representing this evidence is to provide traceability for the origins of the hypotheses, and help stakeholders judge their confidence that the hypothesis is true.

The structure of the evidence entity is:

Element	Description
Description	Text description of the evidence
Old hypothesis	Original hypothesis that is being evolved based on the evidence
New hypothesis	New hypothesis generated by evidence
Studies	Set of studies whose results make up the evidence

Note that the evidence entity may contain information related to multiple studies. Over time, as new studies are performed that affect a hypothesis, this entity must be modified by the researcher to incorporate the new information.

7.4.3 Implication

Implications are consequences of hypotheses that have some meaning for a stakeholder. Any number of stakeholders can be represented in the scheme. As an example, the stakeholders that have been defined in the scheme so far are:

Stakeholder	**Description**
Researcher	Someone who does empirical software engineering research in this domain. A researcher is concerned with methodological issues such as threats to validity.
Professor	Someone who teaches (future) students at the undergraduate and graduate level about HEC. A professor is concerned with teaching programmers without prior domain experience how to effectively write codes for such systems.
Programmer	Someone who writes software that runs on an HEC system. A programmer is concerned with choosing appropriate technologies.
Technologist	Someone who builds tools for use by programmers. A technologist is interested in developing new tools or improving existing ones to improve programmer productivity.
Administrator	Someone who administers an HEC system. An administrator is interested in minimizing costs and maximizing the productivity of a group of scientists, who may have conflicting needs.

7.4.4 Topic

All of the entities (hypotheses, evidence, implications) are indexed by topic, which allows users of the system to browse based on category of interest. The topics are not orthogonal: entities can be classified under multiple topics. The following are examples of topics:

- Programming model
- Effort
- Correctness
- Experience
- Debugging
- Tuning
- Porting to a new machine
- Porting to a new language
- Adoption factors
- Performance-effort tradeoffs
- Productivity factors

7.5 Demonstration of the scheme

In the following sections, we illustrate the application of this scheme. We provide examples from the research of how an initial set of hypotheses was evolved through evidence supported by studies, along with the implications of these hypotheses. The complete set of all hypotheses, evidences, and implications generated by this research is stored in a prototype implementation of the experience base. In each section, an example using different hypothesis types from Table 7.2 is used.

7.6 Initial set of hypotheses

Listed below is a subset of the initial hypotheses in the research. As described in

Table 7.2 initial hypotheses do not stem directly from evidence, but are based on the initial beliefs of the researchers. The examples listed in the following sections will demonstrate how these hypotheses evolved over time.

- **O.1.1** It requires less effort to write parallel programs using a shared memory model (e.g. OpenMP) than a message-passing model (e.g. MPI).
    - Topics: programming model, effort
- **O.1.2** It requires less effort to write parallel programs in a partitioned global address space model (e.g. UPC, CAF) than a message-passing model, but requires more effort than a shared memory model.
    - Topics: programming model, effort
- **O.1.3** It requires less effort to write parallel programs in the PRAM model (e.g. XMT-C) than any of the other parallel programming models.
    - Topics: programming model, effort
- **O.1.4** The effort required to implement parallel programs will depend upon the nature of the communication pattern: the more complex the pattern, the more effort will be saved by using shared memory rather than message-passing.
    - Topics: programming model, effort
- **O.2.1.** Programs written in OpenMP will get better speedup on an SMP system than MPI programs running on the same system.
    - Topics: programming model, performance
- **O.2.2.** Programs written in OpenMP will get better speedup on an SMP system than MPI programs running on a cluster (on the same number of

processors).

    - Topics: programming model, performance

- **O.2.3** Programs written in UPC/CAF and running on a custom HEC machine will get better speedup than MPI programs running on the same system.
    - Topics: programming model, performance
- **O.2.4.** Novices will not be able to get near-linear speedup using MPI on a cluster.
    - Topics: experience, performance
- **O.2.5.** Novices will be able to get near-linear speedup using OpenMP on an SMP system.
    - Topics: experience, performance

While we refer to these here as our "original" list of hypotheses, some of these actually stem directly from the folklore that was collected (see Appendix D). For example, **O.1.1** is a consequence of **F.1**, although it is also partially contradicted by F.5. **O.2.1** is actually contradicted by **F.6**, so it is reasonable to question why we chose this as an initial hypothesis. In fact, **F.6** is one of the more controversial elements of folklore. In retrospect, we may say that **F.12** and **F.13** also contributed to support **O.2.1**: we expected that the students may use many small messages, yielding poor performance, and we knew that the number of processors involved were small (<10).

7.7

7.8 Example: Support for an existing hypothesis

In some cases, new evidence supports an existing hypothesis but does not modify the hypothesis in any way. This is the one situation where evidence does not lead to the creation of a new hypothesis. In this example, we use hypothesis **O.1.1** described in the previous section.

7.8.1 Sources of evidence

- Folklore focus group (Iteration 1)
- folklore survey (Iteration 2)
- informal (email from ICSE 06 reviewer, email from practitioner).

7.8.2 Description of evidence

All of the folklore research provided strong support for O.1.1 One of the initial elements of folklore collected was expressed as: *It's easier to get something working using a shared memory model than message passing.* There was general agreement among practitioners that shared-memory models require less effort than message-passing models: a survey of HEC practitioners at an HPCS meeting in iteration 4 showed that out of 28 respondents, 17 agreed with the statement, 4 disagreed, and the rest either did not respond or indicated that they did not know.

To some members of the community, the idea that shared memory programming requires less effort than message-passing is so apparent that empirical research in this area is unwarranted. Consider the following review from a paper, based on one of the studies described in this dissertation, which was rejected by ICSE 2006 (references added):

It is well-known that shared memory is easier to program than distributed memory (message passing). So well know[n] is this, that numerous attempts exist to overcome the drawbacks of distributed memory. These attempts range from hardware (symmetric multiprocessors), to operating systems (distributed shared memory systems: there are at least a dozen projects about simulating shared memory on a distributed memory) to programming languages (High Performance Fortran [Bokus94], Fortran-D [Fox90], Snyder's ZPL [Chamberlain98], Philippsens's JavaParty [Phillipsen97], and many others). All of this work was done because it was obvious how hard message passing is.

All of the practitioners who took part in the folklore interviews generally agreed that it is more difficult to produce correct code using the message passing model as compared to the shared memory model.

One of the few dissenters from this hypothesis was Brad Chamberlain of Cray, who was not convinced that shared memory was always superior in terms of programmer effort. He stated in an email[11] that: *"I think it really depends. Shared memory codes may have a tendency to be shorter and therefore seem easier to write, but they can be a real hassle to debug when there are subtle synchronization issues".*

7.8.3 New hypotheses

In this case, no new hypotheses are created.

[11] Sent to HPCS Development Time mailing list, 9/21/04

7.8.4 Implications

Technologist

For language developers, the implication here is to favor languages which have shared memory constructs. Since this element of folklore is widely believed by the community, we already see such trends in the development of partitioned global address space languages such as UPC and CAF.

Programmer

For the programmer, if a machine that supports a shared memory model is accessible and has sufficient resources to run the program, then this option should be favored over a message-passing model.

Professor

For the professor who wishes to impart this information to students, the professor should expose students to the different programming models and different communication patterns, so the student can witness firsthand the difference in effort among models, how this difference changes by programming problem, and how difficult it can be to get performance in MPI.

7.9 Example: Refinement of a hypothesis

A hypothesis is refined when evidence from studies provide more detail about a hypothesis. These are classified as "Refinement" hypotheses in our scheme (see Table 7.2). We begin with hypotheses **O.1.1, O.1.2, O.1.3** and **O.1.4** as our initial hypotheses.

7.9.1 Sources of evidence

- Classroom studies (C-S04,D-S04,G-F04,F-S05,C-S05,D-S05,B-F05)

7.9.2 Description of evidence

Through the classroom experiments, we are able to quantitatively evaluate the effect of the programming model on effort, although only for problems that are simple enough to be applicable in classroom assignments, and only using programmers that are not experienced in parallel programming.

In all, there were 13 studies which allowed for direct comparisons between programming models. In each case, MPI was compared to another language. Table 7.3 summarizes the results from the studies, which are described in Section 6.2 .

Table 7.3 Summary of effort studies

Models compared	Studies with stat. sig. results	Mean effort savings
MPI:OpenMP	4/10	37-88%*
MPI:UPC/CAF	0/3	16-48%
MPI:XMT	1/1	50%

* Except for Buffon-Laplace problem

The results of the studies generally supported **O.1.1**, although statistically significant results were not obtained in all cases. Two of the cases where statistically significant results were not obtained involved the Buffon-Laplace needle problem (the only embarrassingly parallel problem examined). In one of these two cases involving Buffon-Laplace, the mean OpenMP effort was more than the mean MPI effort, the only study where this phenomenon was observed.

There was weak support for **O.1.2**. While statistically significant results were not achieved in any of the studies involving UPC or CAF, in each case the mean MPI effort was greater.

There was insufficient data in these studies to draw major conclusions about O.1.3 The single study involving XMT-C showed a statistically significant effort reduction over MPI (50%), which supports both **O.1.3** and **O.1.1** (XMT-C uses a shared-memory model). However, since there were no comparisons to other models, there is no other evidence to support or refute **O.1.3** in these studies.

7.9.3 New hypotheses

The new hypotheses now contain more quantitative details, although their contexts are restricted because evidence is based only on solving certain types of problems. Many more studies were done with OpenMP than with UPC/CAF or XMT-C, so **H.1.1** is broader in scope than **H.1.2**, which is broader in scope than **H.1.3.**

- **H.1.1** OpenMP requires *approximately 35-80%* less effort than MPI, *on small problems that are not embarrassingly parallel, for programmers with little parallel programming experience*.
    - Topics: programming model, effort, experience
- **H.1.2** UPC/CAF requires *approximately 5-35%* less effort than MPI, *for sparse-matrix and cellular automata problems*[12]*, for programmers with little parallel programming experience*.
    - Topics: programming model, effort, experience

[12] e.g. game of life, sharks & fishes

- **H.1.3** XMT-C requires *approximately 50%* less effort than MPI, *for small sparse-matrix problems, for programmers with little parallel programming experience.*
    - Topics: programming model, effort, experience

7.9.4 Implications

Stakeholder: programmer

In cases where the developer has a legitimate choice across parallel programming models, **H.1.1, H.1.2, H.1.3** should help guide the programmer in deciding which model to select by providing some sense of the effort that would be saved by using an alternative to MPI. However, we have little confidence that these results generalized to more experienced programmers and larger programs. In addition, effort is only one factor in a project, which must be balanced against others such as scalability, availability of programming model on target platform, familiarity with base programming language, etc. As Pancake has noted: *With few exceptions, you don't pick the language; it picks you*. [Pancake96].

Stakeholder: technologist

For technology developers, hypotheses **H.1.1, H.1.2, H.1.3** provide a sense of the magnitude of the impact of existing alternatives to MPI on programmer effort. These results can be used as a baseline to form a basis for comparison against newer technologies.

7.10 Example: Explanation of an observed effect

When an unanticipated effect is observed in a study, researchers may generate one or more hypotheses that would explain this effect, to be investigated in future studies. These are classified under "Explanatory" in our scheme (see Table 7.2). Such explanatory hypotheses may not necessarily evolve from any particular hypothesis.

7.10.1 Sources of evidence

- Class project case study (B-S06).

7.10.2 Description of evidence

In one of the class projects, the students attempted to implement a chess-playing program using two different programming models: MPI and Titanium [Yelick98], which is a PGAS (partitioned-global address space language) based on Java. Due to some difficulties described below, the students eventually gave up on Titanium and used OpenMP instead.

The chess program uses a distributed search to determine what move to make next. This algorithm was embarrassingly parallel, and based on the results from the classroom experiments, we would expect the two models to require roughly the same amount of effort to implement the program. In fact, the project members reported that MPI was actually easier than OpenMP for this problem, and that both models were easier than Titanium. In addition to some initial problems with getting the Titanium compiler to work on the target machine, the students had too much difficulty with the Titanium programming model, and abandoned it.

Their chess program relied on non-determinism: a random number generator was used to break ties when deciding between equivalently "good" moves. The students could not determine how to implement this non-determinism in the Titanium model, and eventually gave up. They had no such problems with MPI and OpenMP. Using this project to compare models may not be valid because the students had no prior programming experience with Titanium, whereas they had completed a programming assignment in both MPI and OpenMP. Still, given Titanium's claims of being a more productive language than MPI [Datta05], this one example illustrates obstacles to Titanium adoption.

7.10.3 New hypotheses

The following hypotheses would explain why Titanium was not adopted in this project:

- **H.1.6.** The PGAS model is more difficult than the message-passing model for certain types of problems.
    - Topics: programming model, effort
- **H.1.7.** A minimum level of experience with a PGAS language is necessary before a programmer can use it effectively, even if the programmer has experience with other parallel programming models.
    - Topics: programming model, effort, experience

7.10.4 Implications

Stakeholder: programmer

H.1.6 has implications for both the selection and adoption of PGAS languages such as UPC, CAF and Titanium. Practitioners would benefit from knowing in advance whether a PGAS language would be suitable for their particular problem. **H.1.7** also has implications for practitioners who are considering adopting a PGAS language. As in the classroom projects, programmers may reject a language such as Titanium because of a steeper learning curve, even if it would result in increased productivity in the long run. Practitioners should be willing to invest sufficient effort in their evaluation to learn the concepts of the language before rejecting it.

Stakeholder: researcher

H.1.6 suggests that researchers need to better characterize the space of programming problems so that practitioners have the information they need to select a language. **H.1.7** suggests that researchers who are studying parallel programming languages need to ensure that their subjects have sufficient programming experience with the language under investigation.

Stakeholder: technologist

H.1.7 has implications for developers of PGAS languages. They may need to take some sort of action to reduce the learning curve (e.g. better tutorial documentation) so that users can benefit from the long-term advantage of using a PGAS language, if indeed they are more productive.

Stakeholder: professor

H.1.7 suggests that professors may want to introduce PGAS languages through class assignments if they intend for any students to use these languages in a class project, to allow students to gain some initial experience with the languages.

7.11 Example: Generalization of an existing hypothesis

The scope of a hypothesis can be broadened in the face of additional evidence. These new hypotheses are classified as "Generalization" in our scheme.

7.11.1 Sources of evidence

- Folklore interview (iteration 5) with Richard Kendall of LANL

7.11.2 Description of evidence

Richard Kendall of Los Alamos National Labs made the observation that shared memory programs are also inherently easier to understand as well as being easier for implementing correct code.

7.11.3 New hypotheses

- **H.1.8.** It requires less effort to write *and maintain* parallel programs that use a shared memory model compared to a message passing model.
    - Topics: programming model, effort, maintenance

7.11.4 Implications

Stakeholder: researcher

H.1.8 suggests that the difference in effort between programming models extends into the maintenance phase of the software development process. However, as we have not conducted any studies that compare maintenance activity across languages, we have no basis for estimating the magnitude of the effort savings. Since HEC codes can be very long-lived (decades or more), we would expect maintenance to be a substantial fraction of the total development cost.

Stakeholder: professor

H.1.8 suggests that if a professor wishes to impart information about the differences in development effort across programming models, they may want to give a class assignment or project that involves some sort of maintenance activity, such as adding a feature to an existing parallel program of moderate size.

7.12 Example: Observation of new phenomena

It is often the case when conducting the study that some interesting effect is observed that was not originally being investigated as part of the design of the study. Such effects often warrant further study. This is more common in the more exploratory-oriented, qualitative studies, but can also occur in controlled experiments through correlation analysis. These new hypotheses do not evolve explicitly from an earlier hypothesis.

7.12.1 Sources of evidence

- Case study (ASC-Alliance studies)

7.12.2 Descriptions of evidence

The ASC-Alliance case studies do not provide direct evidence about comparisons of programming models because all of them use MPI exclusively. However, these studies do provide some context about the role of programming model in development effort.

In several of the ASC-Alliance projects, the developers built an abstraction layer on top of MPI. With this layer in place, programmers are not required to write MPI code when adding new features, (assuming the infrastructure supports the desired type of modification). With such an infrastructure in place, we would expect the maintenance effort to be less sensitive to parallel programming model than the initial development effort. Since the classroom studies only capture the initial development effort, then the results of the studies may overestimate the impact of parallel programming model.

The ASC-Alliance case studies also revealed that the effort required to build such an infrastructure atop MPI is substantial. In at least one project, it required years before the code reached at a point where the code was being used for useful science. While there exist publicly-available parallel frameworks that provide a layer of abstraction on top of MPI (e.g. Charm++ [Kale93], POOMA [Oldham02]), there is evidence that these frameworks are difficult to reuse effectively. One of the ASC-Alliance projects at one time used Charm++ but has since abandoned it. An example from outside the ASC-Alliance is the ASCI Blanca project, which encountered many problems that were attributed to its use of the POOMA framework: missing project milestones, cost overruns, and resulting performance that was worse than the original Fortran code [Post03].

One ASC-Alliance interviewee explained that such frameworks require a certain view of the world, and if the particular problem being solved does not fit this view, the amount of effort required to apply the framework is roughly the same as the amount of effort to use MPI, eliminating the advantage of the framework.

7.12.3 New hypotheses

- **H.1.9.** The maintainability of programs that use message-passing models can be improved by using parallel programming frameworks that provide the programmer with a high level of abstraction.
    - Topics: effort, maintenance
- **H.1.10.** Generic parallel programming frameworks are unlikely to reduce the *development* effort for most projects. Since they are difficult to reuse, they must generally be written from scratch.
    - Topics: effort

7.12.4 Implications

Stakeholder: programmer

There are multiple ways to interpret the implications of **H.1.9.** One interpretation is that, for long-lived projects, the differences in effort due to programming model can be reduced by an appropriate software design. Therefore, the effect of programming model on effort should not be a major consideration for such large projects (this interpretation would be in direct contradiction to **H.1.8**). An alternative interpretation is that a programming model such as MPI is so inconvenient for programmers that it forces them to expend considerable effort on an abstraction layer to insulate programmers from future pain. In addition, **H.1.10** implies that reuse cannot be applied to solve this problem. A more "productive" parallel programming model would be one that does not require this sort of abstraction layer. Unfortunately, since we have not examined any mature projects that involved other programming models, it is not clear which interpretation is better supported (assuming the hypothesis is even correct).

Another implication of **H.1.9** is that a mature project would achieve fewer benefits from switching to a new parallel programming model than a new project would from starting with the new model. Given that a mature project would have to make a substantial effort investment to port to the new model, it is unlikely that a mature project would adopt a new parallel programming model, unless there were some additional compelling reasons to do so.

Stakeholder: professor

One implication of **H.1.9** and **H.1.10** is that professors should teach students about the principles of designing a parallel programming framework, and also teach them about the dangers of building "generic" frameworks that are likely to not be used by anyone. Professors may want to focus on teaching how to build a problem-specific framework, perhaps using an environment such as the Common Component

Architecture [Armstrong99].

7.13 Example: Alternative to a hypothesis

Evidence from a study may appear to directly contradict an existing hypothesis. This results in the creation of an "alternative" hypothesis in our scheme, that has a contrary interpretation to the original hypothesis. Such "alternative" hypotheses are often the most interesting ones, since they represent instances where the evidence in a particular context has contradicted our initial beliefs. The hypotheses under consideration here are **O.2.2** and **O.2.5.**

7.13.1 Sources of evidence

- Classroom studies (B-F05)
- Classroom case study (B-F05)

7.13.2 Description of evidence

Classroom studies

In study B-F05, students had to implement the sharks & fishes problem in both MPI and OpenMP. For OpenMP, they ran on an SMP machine, a SunFire 6800. For MPI, they ran on a Linux cluster. In both cases, the programs were run on 8 processors. The differences in speedup are not statistically significant (p=0.78). This directly contradicts hypothesis **O.2.2**. For both models, less than half of the students were able to achieve any performance improvement at all. For the case of OpenMP, this contradicts hypothesis **O.2.5.**

Table 7.4 MPI-OpenMP speedups

	MPI	**OpenMP**
Mean speedup	1.53	1.77
Median speedup	0.94	0.72
N	10	16

Classroom case study

The students that implemented a chess program using both MPI and OpenMP got much better speedup in MPI on a cluster than they did using OpenMP on an SMP. Note that this is an embarrassingly parallel problem, so we would have expected comparable performance. In addition, they were using a Java version of OpenMP (JOMP), so the performance issues may have been attributed to that implementation (the MPI implementation was also Java-based).

New hypotheses

- **H.2.1** Novices will get roughly the same speedups in MPI on a cluster and OpenMP on an SMP for nearest neighbor problems at small processor counts (~8).
    - Topics: programming model, performance, experience
- **H.2.2** Novices many not be able to achieve near-linear speedup on some nearest neighbor and even on embarrassingly parallel problems in OpenMP.
    - Topics: performance, experience

7.13.3 Implications

Stakeholder: researcher

Researchers worry about many threats to validity when conducting controlled experiments, especially in this domain, because the nature of writing small, two-week assignments is perceived to be very different from the nature of writing “real” HEC programs. In addition, the HEC community has traditionally been very concerned with fair comparisons when comparing performance across different machines [Bailey91]. **H.2.1** suggests that concerns about using different machines in a classroom study (often necessary, because clusters don’t run OpenMP) may be overblown. The evidence suggests that factors other than the performance characteristics of the machines dominate in classroom studies, such as the ability of a novice to write an efficient code.

Stakeholder: programmer

H.2.2 has consequences for programmers who wish to try and leverage OpenMP to improve the performance of their program on an SMP system. If the programmers are not sufficiently experienced, then they may not be able to achieve their performance goals, and therefore they may need access to more experienced OpenMP programmers.

Stakeholder: professor

The implication of **H.2.2** is that achieving performance in OpenMP can be more difficult than one would expect. Professors should try to identify what common problems novices have at achieving performance at problems that would seem to be straightforward to parallelize, and teach strategies for avoiding such problems.

7.14 Implementation

As a proof-of-concept, we have built a prototype system that implements the scheme described in this section. This implementation demonstrates that it is possible to coherently organize all of the results from the various studies into one system that can then be navigated by stakeholders. This implementation uses a relational database to store the different entities, and provides a web-based user interface to navigate through the topics, hypotheses, evidences, and implications through hyperlinks. This implementation can be found at http://care.cs.umd.edu:8080/hpcs/expbase.

Chapter 8 Conclusion

8.1 Summary of contributions

The contributions of this dissertation can be divided up into two categories: those that advance knowledge of the state of the practice in the domain of high-end computing, and those that advance research methodology in empirical software engineering.

8.1.1 Domain-related contributions

Quantifying impact of parallel programming model.

One of the major contributions of this work is the quantification of the impact of parallel programming model on programmer effort. While there were strong opinions about the effects of parallel programming model, no previous attempts to quantify these differences had been made. This contribution provides a starting point for reasoning about tradeoffs between models, and provides benchmarks for future languages.

Identifying productivity issues

The research began by focusing on programming models. As the research proceeded, several other factors were identified as having an impact on productivity. Some relate directly to the software development process, such as effort to port the code to a new platform, interacting with the job queue, and verifying the software. Others are not directly related to the software development itself but potentially involve significant amounts of developer time (if the developers are the users), such as the time required to develop meshes for input, and the effort involved in validating the software.

Making folklore explicit

One contribution is the eliciting and recording the tacit knowledge of HEC practitioners, as there have been no previous attempts to capture this knowledge in a formal and systematic way.

Development of infrastructure for conducting further research

With the development of the experimental package and toolset, HEC professors can run their own studies without direct support from HEC researchers.

Information about student development habits

Through the results of the classroom studies, professors now have information about student development habits when writing parallel programs which they can use to help future students develop software better, by incorporating the material into their courses. Future studies are now planned to feed back knowledge about defects obtained from analysis of the data to try and improve programmer's abilities to conduct these studies.

Spreading the gospel of empiricism

One indirect contribution of this dissertation is the promotion of the idea of empirical software engineering research in the HEC community. At the completion of this dissertation, thirteen professors have participated in these studies (Dr. Jeffrey Hollingsworth at UMD, Dr. Alan Sussman at UMD, Dr. Uzi Vishkin at UMD, Dr. Alan Edelman at MIT, Dr. Allan Snavely at UCSD, Dr. Henri Casanova at U. Hawaii, Dr. Glenn Luecke at Iowa State, Dr. Ed Luke at Mississippi State, Dr. Mary Hall at USC, Dr. Jacqueline Chamé at USC, Dr. John Gilbert at UCSB, and Dr. Aiichiro

Nakano at USC), and many have participated in multiple studies. In addition, a paper generated from this dissertation work won the best student paper award at SC'05. If nothing else was accomplished but exposing this community to the value of conducting human-subject research, then this work would still be considered a success by the author.

Repository of hypotheses

We began this research with about 10 hypotheses that relate to high-end computing recorded explicitly. At the termination of this dissertation, we have over 50 hypotheses.

8.1.2 Methodological contributions

Methodology for building knowledge iterative through opportunistic studies

The main methodological contribution of the dissertation is the proposal for the iterative, opportunistic approach that leverages a variety of different types of empirical studies to build a broad base of knowledge.

Effort measurement and evaluation

Another methodological contribution generated by this research is a validated method for estimating programming effort based on data collected automatically from the programmer's environment using instrumentation software.

Organizational scheme for experimental results

The organizational scheme and the associated methodology for storing, updating, and navigating through the collected studies is the third methodological contribution of this research. This methodology is general enough that it should be applicable to building knowledge in any other domain of software engineering.

8.2 Lessons learned

Here we describe some general lessons learned from conducting this research.

8.2.1 Asking the right questions

HEC practitioners are an enormous source of knowledge. Many of them have years and years of experience developing codes in this domain, and they are eager to discuss their experiences. One of the largest challenges for the researcher is simply determining which are the right questions to ask. This reinforces the importance of an iterative approach: as the researcher builds knowledge in the domain, new issues come to light and can be pursued.

8.2.2 Speaking the language

When conducting interviews for case studies and folklore gathering, there were sometimes mismatches in how we used a term and how the interviewees understood what the term meant. For example, when we used the term “tuning”, we had in mind the idea of modifying a code to improve the performance. However, practitioners sometimes interpreted this as machine-specific optimizations. Another example is that of “defects”. In software engineering terminology, a defect applies to a fault in any artifact (e.g. code, design, requirements). However, the practitioners interpreted “defects” only as problems in the code. They drew a distinction between “software defects” (defects in implementing the algorithm) and “algorithmic defects” (defects in the algorithms themselves, in the sense that they do not have the desired numerical properties for the project).

Another miscommunication occurred when discussing the distinctions between writing a correct parallel program and achieving good performance. From the folklore, and based on the classroom studies, we had observed many instances of cases where a parallel program was "correct" in the sense that it produced the correct answer, but its performance was poor. When we tried to discuss this issue in the ASC-Alliance interviews, they were confused at the distinction, which was not meaningful to them. They seemed to know how to implement the code so that no additional step of "tuning" or "optimizing" was required.

This suggests that an ethnographic approach may be necessary to pursue research in this domain, so that the researcher can become immersed in the culture and avoid the problems with different uses of language and unshared sets of assumptions. But because of time constraints and the fact that almost every group has its own models and definitions, studying the ethnography is an on-going process in this study.

8.2.3 Challenges to running SE studies in HEC courses

HEC courses provided us with a wonderful opportunity to run controlled experiments. However, this environment also presented several challenges.

Identifying representative problems

A common criticism of these studies is that the problems were not representative of the types of problems that HEC practitioners actually solved. However, there are various obstacles to choosing more relevant problems. The students typically have two weeks to solve the problem, so it must be small enough to be solvable. In addition, these courses generally do not require a background in numerical computing, which limits the mathematical complexity of the problem.

Performance incentive

The incentives for students working on the class assignments to achieve certain levels of performance are very different from the incentives of practitioners. The students are given a particular problem to solve, and possibly some sample inputs and outputs, and are asked to implement the program in parallel, trying to get it to run as fast as they can. By contrast, practitioners simply want to complete a production run on the resources they have available. They do not care about "speedup", only about getting the job done.

For the students to be given similar incentives, they would have to be given a problem where the input provided was so large that they would be forced to write a parallel problem to process the output in the appropriate amount of time. However, it is difficult to introduce this type of problem in a classroom study.

Pedagogy versus quality of the study

The priority of the courses must always be pedagogy over the quality of the study. This places limits on the types of controls that can be put on the study, since a subset of the students cannot be given an unfair advantage (e.g. exposed to a useful treatment) to benefit the study. For example, having the students add a feature to an existing code would be more representative of the development activities in this domain. However, it would deny the students the opportunity to design the parallel algorithms from the ground up.

8.3 Future avenues of research

8.3.1 Domain-related

Adoption issues

The evaluation aspects of this research focused on measuring the impact of programming model on aspects of productivity. However, even if a technology can be empirically demonstrated to improve productivity, this does not guarantee that it will be adopted. In particular, the HPC community tends to be conservative in technology adoption, favoring stable, proven technologies (e.g. Fortran, C). One future avenue of research is to identify the factors that result in new software technologies being accepted in this domain.

Understanding tools

In this dissertation, the focus was largely on parallel programming languages because of the community's belief that this is an avenue where advancement is needed to improve productivity. However, there are many different types of software development tools for the HPC domain (e.g. debuggers, performance analyzers), and understanding how these tools impact productivity should be pursued as well, especially since such tools often have fewer obstacles to adoption when compared with new programming languages.

Minimizing the gap between experts and novices

The perception in the community is that there is a wide gulf between the ability of "novice" and "expert" HPC programmers to achieve good performance. Indeed, this was a frequent criticism of the classroom studies. One future avenue of research is to attempt to capture the knowledge of experts in such a way that it can be packaged and disseminated to novices to increase their productivity. Some of this should be with respect to professor's classes

8.3.2 Methodological

Better decision-support interface for practitioners

The prototype implementation for the experience base allows practitioners to browse the entire base or search by topic. It would be more useful for practitioners if the system was designed specifically for decision-support: to locate evidence to support certain types of decisions. One possibility would be to use an ontology to encode the information stored in the experience base and allow practitioners to make appropriate queries.

Confidence ratings for hypotheses

The current system for storing hypotheses does not directly encode the degree of support for a hypothesis. While a user can navigate through the system to learn about the evidence that supports the hypothesis, there is no sense of a "confidence rating" for a particular hypothesis. Developing this type of confidence rating of a hypothesis in a particular context is a potential future avenue of research.

Workflow

The results of the controlled experiments that involved effort did not distinguish between the various development activities. One potential avenue of research is to automatically identify the time spent in different development activities to construct an “operational profile” of HPC developers. The more we understand what programmers are actually doing, the better we can help them.

Appendix A Details about individual iterations

Table A.1 Iterations overview

Iteration	Date	Studies conducted
1	Fall 2003	Classroom: A-F03
2	Spring 2004	Classroom: B-S04, C-S04, D-S04, E-S04, F-S04 Other: survey
3	Fall 2004	Classroom: G-F04, H-F04 Observational: Obs-1, Obs-2 Folklore: focus group
4	Spring 2005	Classroom: C-S05, D-S05, E-S05, I-S05 Folklore: survey
5	Summer 2005	Observational: Obs-3 Folklore: Interviews
6	Fall 2005	Classroom: B-F05, H-F05 Case studies: ASC-1,ASC-2,ASC-3 Class case studies: B-F05-C1, B-F05-C2, B-F05-C3, B-F05-C4, B-F05-C5

Table A.2 Classroom programming problems

Study	Problem	Models	Given serial?
A-F03	Game of life	MPI	No
A-F03	SWIM	OpenMP	Yes
B-S04	Game of life	MPI	No
D-S04	Buffon-Laplace	MPI, OpenMP, MATLAB*P	No
C-S04	Randomized selection	XMT-C	No
D-S04	Grid of resistors	MPI, OpenMP, MATLAB*P	Yes
D-S04	Laplace’s equation	MPI, OpenMP, MATLAB*P	No
E-S04	Buffon-Laplace	MPI, OpenMP, MATLAB*P	No
E-S04	Sorting	MPI	No
E-S04	Game of life	MPI, OpenMP	No
E-S04	Sparse-matvec	MPI	No
F-S04	LU decomp	OpenMP	Yes

G-F04	Parallel sum	MPI, OpenMP	No
G-F04	Matrix multiply	MPI, OpenMP	No
G-F04	Sharks & fishes	MPI, OpenMP	No
G-F04	Sharks & fishes	Co-Array Fortran	Yes
H-F04	Quantum dynamics	MPI	Yes
G-F04	Parallel sum	MPI, OpenMP	No
C-S05	Sparse-matvec	XMT-C	No
C-S05	Randomized selection	XMT-C	No
C-S05	Breadth-first search	XMT-C	No
D-S05	Matrix power via prefix	MPI, OpenMP	No
D-S05	Sparse conjugate-gradient	MPI+MATLAB*P	No
E-S05	Game of life	MP, OpenMP, UPC, Co-Array Fortran, MATLAB*P	No
E-S05	Sparse conjugate-gradient	MPI, OpenMP, UPC, CAF, MATLAB*P	No
I-S05	Sorting	MPI, OpenMP	No
B-F05	Sharks & fishes	MPI, OpenMP	No
H-F05	Molecular Dynamics	Hybrid MPI+OpenMP	Yes (given MPI)

Table A.3 Observational study problems

Study	**Problem**	**Models**	**Given serial?**
Obs-1	Buffon-Laplace	MPI	No
Obs-2	Game of life	MPI	No
Obs-3	Game of life	MPI	No

Appendix B Programming problems

This appendix contains brief descriptions of the different parallel programming problems that were given throughout the course of the studies.

B.1 Buffon-Laplace needle problem

Imagine that a needle of length *l* is dropped onto a floor with a grid of equally spaced parallel lines, distance *a* and *b* apart. The probability that the needle will land on at least one line is given by:

$$P(l,a,b) = \frac{2l(a+b) - l^2}{\pi ab}$$

The programming task is to estimate the value of pi by simulating needle drops.

B.2 Game of life

The game of life simulates simple cellular automata. The game is played on a rectangular board containing cells. At the start, some of the cells are occupied, the rest are empty. The game consists of constructing successive generations of the board. The rules for constructing the next generation from the previous one are:

1. death: cells with 0,1,4,5,6,7, or 8 neighbors die (0,1 of loneliness and 4-8 of over population)

2. survival: cells with 2 or 3 neighbors survive to the next generation.

3. birth: an unoccupied cell with 3 neighbors becomes occupied in the next generation.

B.3 SWIM

SWIM is a sequential Fortran shallow water model code that is part of the SPEC Benchmark suite. The task is to parallelize the given code.

B.4 Randomized selection

Given a set of N integer elements, and an integer k ($1 \leq k \leq N$), find the *k*th smallest element in the set in expected linear time.

B.5 Grid of resistors

The problem is to compute the voltages and the effective resistance of a 2n+1 by 2n+2 grid of 1 ohm resistors if a battery is connected to the two center points. This is a discrete version of finding the lines of force using iron filings for a magnet. The solution is to be computed using successive overrelaxation. The nodes are divided in half into red nodes and black nodes. During the first pass, the red nodes obtain the voltages as a weighted average of their original voltage, the input (if any) and the four surrounding black nodes. During the second pass, the black nodes obtain voltages from the four surrounding red nodes. The process converges in the limit to the correct answer for the finite grid.

B.6 Laplace's equation

Solve Laplace's equation in two dimensions. Use a simple discretization of Laplace's equation on a rectangular grid. The solution to the discrete equation has the temperature at each grid point to be the average of its four neighbors in each direction. The boundary conditions are specified one grid point beyond the edge of the grid.

B.7 Sorting

Sort a set of N elements.

B.8 Sparse matvec

Multiply a sparse matrix with a dense vector.

B.9 LU decomposition

In LU decomposition a square matrix is factored into two matrices L and U, where L is lower triangular with ones on its diagonal, and U is upper diagonal.

B.10 Parallel sum

Sum a list of elements.

B.11 Matrix multiply

Multiply two (dense) matrices.

B.12 Sharks & fishes

(Note: this is one example of the rules for the sharks & fishes problem, the rules varied from one class to another).

Imagine the ocean divided into a square 2D-grid:

- Each grid cell can be empty or have a fish or a shark
- Grid initially populated with fishes and sharks in a random manner
- Population evolves over discrete time steps according to certain rules

Rules for Fish

- At each time step, a fish tries to move to a neighboring empty cell. If not empty, it stays. Fish move up or down, left or right, but not diagonal (like Rooks not Bishops)
- If a fish reaches a breeding age, when it moves, it breeds, leaving behind a fish of age 0. Fish cannot breed if it doesn't move.
- Fish never starve

Rules for Sharks

- At each time step, if one of the neighboring cells has a fish, the shark moves to that cell eating the fish. If not and if one of the neighboring cells is empty, the shark moves there. Otherwise, it stays. Sharks move up or down, left or right, but not diagonal (like Rooks not Bishops)
- If a shark reaches a breeding age, when it moves, it breeds, leaving behind a shark of age 0. A shark cannot breed if it doesn't move.
- Sharks eat only fish. If a shark reaches a starvation age (time steps since last eaten), it dies.

B.13 Quantum dynamics

Simulate an electron wave function in one dimension. The task is to parallelize a given serial implementation.

B.14 Breadth-first search

Given a connected undirected graph, visit the vertices in "breadth-first search" order: for any given vertex, visit all outgoing edges of the vertex's predecessor before any outgoing edges of the vertex.

B.15 Matrix power via prefix

Given an NxN matrix A, compute all its powers (up to K). That is, you want A, A^2, ..., A^K.

B.16 Sparse conjugate-gradient

Solve a sparse linear system using the conjugate-gradient method.

B.17 Molecular dynamics

Simulate the interaction of a system particles interacting with the Lennard-Jones potential. The task is to parallelize a given serial implementation.

Appendix C Data from classroom studies

This appendix contains effort and defect data captures from the classroom assignments. Two types of instrumented effort data are presented: *reported effort* and *instrumented effort*. *Reported effort* data is based on effort logs submitted by the students. *Instrumented effort* is an estimate of effort using captured data, as described in Chapter 5. In many cases there is no *instrumented effort* data. In these cases, the most likely event is that the students did not enable instrumentation in their accounts. In all cases, the units of effort are hours.

C.1A-F03

Table C.1 A-F03 effort: game of life

Subject	Model	Reported effort	Instrumented effort
1	mpi		5.1
2	mpi	59	12.0
3	mpi	24	0.2
4	mpi	25	10.0
5	mpi	12.5	1.8
7	mpi	7	25.0
8	mpi	11.5	7.7
9	mpi	11	4.1
10	mpi		0.2
11	mpi	17.75	4.6
12	mpi	28	16.0
13	mpi	10.5	13.4
14	mpi	21	1.7
15	mpi	9	15.3
16	mpi	17	20.7
18	mpi	43	37.8

Table C.2 A-F03 effort: SWIM

Subject	Model	Instrumented effort
1	openmp	5.7
2	openmp	14.6
3	openmp	1.8
4	openmp	2.0
5	openmp	4.9
6	openmp	6.5
7	openmp	2.9
8	openmp	3.2
9	openmp	2.9
11	openmp	1.4
12	openmp	8.1
13	openmp	3.8
14	openmp	1.6
15	openmp	2.8
16	openmp	1.8
18	openmp	8.5

Note: We did not collect reported effort from A-F03 SWIM.

C.2 B-S04

Table C.3 B-S05 effort: game of life

Subject	Model	Instrumented effort
1	mpi	1.0
2	mpi	1.9
3	mpi	4.9
4	mpi	11.8
5	mpi	3.0
6	mpi	0.1
7	mpi	9.9
8	mpi	4.4
9	mpi	14.2
10	mpi	15.3
11	mpi	3.9

Note: We did not collected reported effort from B-S04, due to an oversight.

C.3 C-S04

Note: We did not analyze data from C-S04 because of problems that occurred during the study. The students were developing with an experimental prototype compiler for the XMT-C language, and they ran into many problems because of compilers issues, which increased their effort spent in development for reasons separate from the language itself.

C.4 D-S04

Table C.4 D-S04 effort: Buffon-Laplace

Subject	Model	Instrumented effort	Reported effort
1	mpi	0.4	2
1	openmp	0.4	1
1	matlab*p		3
4	mpi	1.1	
4	openmp	0.9	
4	matlab*p		1
5	mpi	0.5	
5	openmp	0.2	
8	mpi	4.6	
8	openmp	8.5	
10	mpi	0.1	
10	openmp	0.9	
14	mpi	0.7	
14	openmp	0.8	
15	mpi	0.5	
15	openmp	0.7	
17	mpi	0.6	8
17	openmp	0.5	
20	mpi	1.2	
20	openmp	0.7	
21	matlab*p		3
25	mpi	0.6	
25	openmp	0.6	
26	mpi	19.4	
26	openmp	8.2	

Notes:

- We did not have instrumentation for the MATLAB environment, so there is no instrumented effort for MATLAB*P for any of the assignments.
- Due to a defect in the script which processed the reported effort data, for subjects 14 and 26 the assignment associated with the effort was not known: Subject 14 reported 8 hours of MPI, 1 hour of OpenMP and 4 hours of MATLAB*P, and subject 26 reported 6 hours of MPI. This data is either associated with the Buffon-Laplace problem or the grid of resistors problem.

Table C.5 D-S04 effort: grid of resistors

Subject	Model	Instrumented effort	Reported effort
1	mpi	2.2	6
1	openmp	0.5	7
1	matlab*p		5
4	mpi	4.6	
4	matlab*p		2
8	mpi	7.0	
8	openmp	0.7	
10	mpi	0.3	4
10	openmp	0.6	
14	mpi	0.9	
14	openmp	1.4	
14	matlab*p		2
15	mpi	2.2	
15	openmp	3.4	
17	mpi	0.2	
17	openmp	2.1	
17	matlab*p		10
20	mpi	2.5	
20	openmp	2.6	
25	mpi		
25	openmp	0.3	
26	mpi	7.4	4
26	openmp	1.1	

Table C.6 D-S04 effort: Laplace's equation

Subject	Model	Instrumented effort	Reported effort
1	mpi		2
1	openmp		4
4	mpi	4.5	
4	matlab*p		1.5
17	mpi	4.3	
26	mpi	17.9	

C.5 E-S04

Table C.7 E-S04 effort: Buffon-Laplace

Subject	Model	Instrumented effort	Reported effort
1	mpi	0.3	
1	openmp	0.9	
2	mpi	0.2	
2	openmp	1.1	
3	mpi	0.4	
3	openmp	0.6	
4	mpi	1.6	
4	openmp	1.6	
5	mpi	1.8	
5	openmp	1.0	
6	mpi	1.1	
6	openmp	0.5	
7	mpi	1.1	
7	openmp	3.4	
8	mpi	2.1	
8	openmp	3.0	
9	mpi	1.0	
9	openmp	0.8	
10	mpi	0.2	
10	openmp	0.7	
12	mpi	4.5	
12	openmp	3.1	
13	mpi	2.1	
13	openmp	2.5	

14	mpi	1.0	
14	openmp	2.4	
15	mpi	2.4	3.0
15	openmp	2.2	3.0
15	matlab*p		4.0
16	mpi	0.3	
16	openmp	1.8	
17	mpi	1.0	
17	openmp	9.1	
18	mpi	1.8	
18	openmp	3.4	
19	mpi	1.3	
19	openmp	2.7	
20	mpi	0.4	
20	openmp	5.2	5.0
21	mpi	3.0	
21	openmp	3.4	

Table C.8 E-S04 effort: sorting

Subject	Model	Instrumented effort	Reported effort
1	mpi	16.1	
2	mpi	0.5	
3	mpi	7.8	
4	mpi	6.7	
5	mpi	3.9	
7	mpi	14.3	10
8	mpi	15.1	14
10	mpi	8.6	
12	mpi	22.7	
13	mpi	8.2	
14	mpi	0.1	
15	mpi	15.4	10
16	mpi	8.23	
17	mpi	9.1	
18	mpi	54.3	43
20	mpi	16.8	
21	mpi	27.2	

Table C.9 E-S04 effort: game of life

	MPI		OpenMP	
Subject	Instrumented effort	Reported effort	Instrumented effort	Reported effort
1	12.1	21.0	2.8	
3	1.3		7.5	
4	0.5	4.0	0.5	4.0
6	0.8		0.8	
7	13.5		5.3	4.0
8	10.5	13.0	4.8	3.0
10	5.0	5.0	2.9	5.0
12	20.9	22.0	3.3	6.0
13	5.3	5.0	3.6	20.0
14	4.1	5.0	4.7	3.0
15	9.7	12.0	1.4	5.0
16	3.1	19.5	0.5	5.0
17	5.1	18.0	3.9	
18	31.5	26.5	13.5	
20	13.8	24.0	9.3	30.0
21		4.0	4.9	18.0

Table C.10 E-S04 effort: sparse matrix-vector multiply

Subject	Model	Instrumented effort	Reported effort
1	mpi	5.7	13.0
3	mpi	6.7	
4	mpi	6.0	14.0
6	mpi	2.1	3.0
7	mpi	18.7	
8	mpi	11.2	16.0
10	mpi	7.5	2.5
12	mpi	8.7	22.0
13	mpi	13.9	32.0
14	mpi	8.8	4.0
15	mpi	4.2	6.0
16	mpi	12.9	
17	mpi	6.9	20.0
18	mpi	13.4	
20	mpi	11.5	44.0
21	mpi	5.1	8.0

C.6 F-S04

Table C.11 F-S04 effort: LU decomposition

Subject	Model	Instrumented effort	Reported effort
1	openmp	13.4	28.5
2	openmp	7.9	8.5
3	openmp	16.4	1.0
4	openmp	17.7	26.0
5	openmp	25.0	1.0
6	openmp	13.7	12.0
7	openmp	13.5	6.0
8	openmp	21.1	1.0

Note: this was a "tuning" problem, to increase the performance of a given Fortran program as much as possible. While there is a parallel component (using OpenMP), much of the optimization in the assignment was to the serial aspects of the code.

C.7 G-F04

Table C.12 G-F04 effort: Dense matrix multiply

	MPI		OpenMP	
Subject	Instrumented effort	Reported effort	Instrumented effort	Reported effort
1	3.1	7.9	2.1	2.7
2				
4				
5				
6	4.1		0.1	
8	11.2	16.0	0.2	
9	5.0	4.1	2.5	2.7
10	36.5	43.9	3.0	6.8
11	11.9		4.2	
14	8.9	10.5	4.2	1.5
15	7.5	14.0	1.7	2.7
16	11.0	24.8	2.5	2.3
17	1.2	15.7	2.9	4.1

Table C.13 G-F04 effort: sharks & fishes

	MPI		OpenMP		Co-Array Fortran	
Subject	Instrumented effort	Reported effort	Instrumented effort	Reported effort	Instrumented effort	Reported effort
1		9.5	3.6	4.8	3.8	
2			1.5			
4		20.3		2.7		
6					0.1	
8	9.6					
9	15.7	15.4	2.1	1.4	14.1	9.8
10			10.5			
11	0.8		1.4			
14	6.3		9.2			19.0
15	12.6	15.0	2.6	6.0		
16	4.5	15.0	9.2	20.0		14.0
17	0.5	17.3	0.5	4.7		9.8

C.8 H-F04

Table C.14 H-F04 defects: quantum dynamics

Subject	**Description**	**Reason**	**Effect**	**Time to fix (minutes)**
01	Used '-' instead of '+' in one of the conditions	typo	incorrect output	30
01	Standard input error	Only one computer read the input. Others not. Program freeze at runtime.	incorrect file/read-in error	150
02	did not include "mpi.h" in ???	forgot about it	compiler error	3
02	declared "status" as int	didn't realize that it was mpi status not a flag	compiler error	2
02	wrote buf[i]=psi[i][0] again	did cut and paste and did not change the values	the program gave faulty o/p	60
02	Had to hard code the i/p file name in the program	Did not know that the program would not work when i/p file is given in command line	The program would hang in between	210
03	Used &argc instead of &argv	Typo error	Not defined	1

03	Used nproc and nprocs, but defined only nprocs	copy, paste of code	Not defined	15
03	gpsiq variable not defined	Typo error	Not defined	15
03	gekin not defined	Typo error	Not defined	7
03	without initializing MPI_Init() called the other function having parallel functions	placed the code at the wrong place.	invalid call to the procedures	4
03	improper checking of the if conditions with myid variable	logical error	no output	30
03	didn't check for myid==0 before printing	overlooked the requirement	no output	20
04	Get wrong integer value because didn't initialize it	didn't initialize that integer with 0	Get very weird value	10
04	Array out of bound	copy code from another section without carefully revise	program crashed	15
04	Get Integer Divion instead of double division	didn't read the manual (help)	get very weird value	30
05	Forgot to add math.h	careless	Program hung	10
05	Missing {	Careless/too crowded code	Program error	2
06	Used wrong variable	wrong understanding of the concept	incorrect output	60
06	only gave input to one process	copy-pasted from another code. didn't know need to change it	program hung, incorrect output	20
07	node down in MPI world	a node down and not response	program executes but did not return results	5
07	Aggregated result from all MPI nodes gave incorrect result	local variables in each node did not get initialized	incorrect output	40
07	Result incorrect when run a dependent program	did not use block send/receive	program not terminated	10
08	Used variable before defining it	carelessnes	compile error	2
08	Used "npros" instead of "nprocs"	typo	compile error	2
08	Used "my_parity" instead of "myparity"	typo	compile error	2

08	Forgot to add condition for fprintf	write to one file from multiple processors	result error	15
08	Send twice and receive twice	may cause buffer overflow	result error	15
08	")" was missing	careless	compile error	2

C.9 C-S05

Table C.15 C-S04 effort: sparse matrix-vector multiply

Subject	Model	Instrumented effort	Reported effort
1	xmtc	1.4	
2	xmtc	5.6	5.6
3	xmtc	0.1	
4	xmtc	2.9	6.4
6	xmtc	2.6	3.8
7	xmtc	5.6	10.0
8	xmtc	3.9	6.3
10	xmtc	3.2	6.2
11	xmtc	4.6	7.0
12	xmtc	3.4	4.8
13	xmtc	15.1	13.6
14	xmtc	6.9	8.5

C.10 D-S05

Table C.16 D-S05 effort: matrix power via prefix

	Instrumented effort		
Subject	MPI	OpenMP	Model used first
1	13.3	4.1	openmp
3	9.6	4.2	mpi
4	15.8	5.0	openmp
5	3.9	1.5	mpi
7	2.9	6.4	openmp
8	4.1	1.7	openmp

We used random assignment into groups to avoid the problem of everyone solving the problem with MPI and OpenMP in the same order. *Model used first* shows which model the students were assigned to solve first. This assignment was determined by the letter of the person's last name. The unequal distribution is because of mortality (e.g. dropping the course).

We did not obtain reported effort data for this class. Effort forms were distributed at the beginning of the term but none were returned.

C.11 E-S05

Table C.17 E-S05 effort: game of life

	Instrumented effort (hours)				
Subject	MPI	OpenMP	MATLAB*P	UPC	Assigned model
1	20.7				mpi
2	9.2			9.9	upc
5	50.8			14.9	upc
7	20.2	6.2			mpi
8		6.3		16.5	upc
9		0.9		2.0	upc
10	37.5		4.9		mpi
12	3.0				mpi
13		3.9		8.7	upc
14	6.2			9.7	mpi
16		0.7			upc
22	7.6	3.3			mpi
23	7.9				upc

Table C.18 E-S05 effort: sparse conjugate-gradient

Subject	MPI	OpenMP	UPC	Co-Array Fortran	Assigned model
1	28.4		22.6		upc
5	47.9		16.8		mpi
7	5.1		28.1		upc
8	29.5		12.2		mpi
9	6.0	6.1			mpi
11	16.5				mpi
12	0.5				caf
16	18.9	7.1			mpi
18	21.3			0.4	mpi
19		0.3	27.0		upc
22	11.8		4.3		upc
23	30.1	8.3			mpi

Students were randomly assigned a programming model (MPI or UPC/Co-Array Fortran) and were able to choose any other model (MPI, UPC/Co-Array Fortran, OpenMP, MATLAB*P). For the game of life problem, One student chose Co-Array Fortran but the instrumentation did not capture any data.

We did not obtain reported effort data for this class. Effort forms were distributed at the beginning of the term but none were returned

C.12 I-S05

Table C.19 I-S05 defects: sorting

Subject	Model	Description	Time to fix (minutes)
01	openmp	Array S was displaying incorrect value	20
01	openmp	Value of Q not proper	40
01	openmp	Wrong value of variable nt	30
01	openmp	Displaying wrong value of variable nt	25
01	openmp	Divide by zero error	10
01	openmp	Missed putting the variable Q in the shared section	15
01	openmp	Variable nt used instead of lastthread	10
01	openmp	Missed putting the variable R in the shared section	10

02	openmp	Off by one error in loop condition	15
02	openmp	Incorrect bounds on parallel-serial transition	45
02	openmp	Global Rearrangement implemented incorrectly	?
02	openmp	Race conditions between cout statements	10
02	openmp	#pragma omp barrier was waiting on threads that had already terminated	120
02	openmp	#pragma omp for was ""stealing"" #pragma omp barriers	180
02	openmp	Global Rearrangement implemented incorrectly	45
02	openmp	Was storing array indices as ints for large arrays.	5
02	openmp	Function parameters and return values were still ints	5
02	openmp	Thread Partitioning implemented incorrectly	20
02	openmp	Global Rearrangement implemented incorrectly	5
02	openmp	Race conditions?	?
03	openmp	incorrect serial sorting on implementation 1	30
03	openmp	parallel algorithm will not function properly	not fixed
03	openmp	incorrect serial sorting on implementation 2	20
03	openmp	segmentation fault during run	40
03	openmp	bus error	not fixed
03	openmp	segmentation fault during run	5
03	openmp	segmentation fault during run	2
03	openmp	bus error	not fixed
03	openmp	bus error	not fixed
03	openmp	incorrect serial sorting on implementation 3	15
03	openmp	incorrect serial sorting on implementation 3	2
03	openmp	bus error	240
04	openmp	Program output was unreadable	1
04	openmp	Overlap of sections within a partition	10
04	openmp	Accessing arrays out of bounds	45
04	openmp	Probably going out of bounds again.	15
04	openmp	Successive iterations attempted to sort the initial list.	15
04	openmp	Successive iterations were not sorting the right partition ranges.	45
04	openmp	Some partitions don't get a processor	10
04	openmp	Problems sorting a partition of size 1.	15
04	openmp	The list wasn't being consistent	30
04	openmp	The list had many empty slots (=0)	15
04	openmp	A processor wasn't always assigned a section to work on.	75
04	openmp	Sublists weren't sorting right	180
05	openmp	segmentation fault	?
05	openmp	race over condition	70
05	openmp	implemented barrier at wrong place	?
05	openmp	incorrect loop for pivot	20
05	openmp	incorrect storage of the variable	20
05	openmp	defined pivot interger in main and double in subroutine	?
05	openmp	was passing wrong variable through subroutine	?
06	mpi	Tried to access first element of array as pivot[1] rather than pivot[0]	10
06	mpi	Loop accessed outside of array bounds	20
06	mpi	Recursive creation of statically allocated arrays caused segmentation violation	45

06	mpi	Processors not receving intended data during global rearragement	90
06	mpi	MPI error due to incorrect arguments to MPI_Gather()	10
06	mpi	Chose last element in array as the pivot in serial quicksort algorithm rather than randomly selecting pivot	20
07	mpi	Nlocal was defined equal for all the processors	20
07	mpi	Did MPI_scatter on processors except 0	10
07	mpi	MPI_Allgatherv()	180
08	mpi	tried to send and receive more than one element from mpi scan	20
08	mpi	error in formula to determine how processors are assigned to the subarrays	42
08	mpi	allocated array of incorrect size to hold received elements	8
08	mpi	mpi_alltoallv requires each process know what to receive, using it without this causes arrays to receive nothing	240
08	mpi	variable was not initialized before being used, resulting in garbage value	2
08	mpi	when sending values to other processors, offsets were not computed	14
08	mpi	all to all personalized was only implemented to work in cases where the processors were a power of 2	120
08	mpi	when using process groups in mpi, group references are not pointers	10
08	mpi	one to all broadcast was sending wrong value for lower array size	30
08	mpi	invalid rank used for all to all personalized	25
08	mpi	tried to access array with old rank of processor, since reassigned	5
09	mpi	Type problem	3
09	mpi	Ceil	5
09	mpi	new ranks	15
09	mpi	cast	5
09	mpi	wrong array passed	5
09	mpi	fence post	10
09	mpi	Invalid rank	60
09	mpi	delete []	120
09	mpi	heap over flow	60
09	mpi	long int needed	30
09	mpi	barrier needed before rand	10
09	mpi	delete []	5
09	mpi	clamping	5
09	mpi	new ranks	20
09	mpi	recursion	5
10	mpi	Wrong value of s after partition.	5
10	mpi	Wrong value of last1for each processor.	15
10	mpi	Wrong value of last1for each processor.	1
10	mpi	Use of MPI_Irecv instead of MPI_Recv.	30
10	mpi	Wrong value of s after partition.	15
10	mpi	Wrong size of array a.	45
10	mpi	Recursion was not possible.	20
10	mpi	New ranks for processors in second group were not correctly calculated.	15
10	mpi	Function get_timer was called thrice.	5
10	mpi	Variable myId was used instead of myId1.	10
11	mpi	Sent wrong number of values for pivot calculation.	5

11	mpi	Array index calculated incorrectly	20
11	mpi	Calculated n for each process incorrectly	20
11	mpi	Calculated distribution of processors incorrectly	10
11	mpi	Used recv to receive broadcast instead of Bcast	30
11	mpi	Used Alltoall instead of Allgather	30
11	mpi	Incorrect array indexing	60
11	mpi	Program crashing (probably due to incorrect indexing)	300
12	mpi	condition in while loop not correct	20
12	mpi	should be ""<="" but used ""<""	?
12	mpi	code modification not consistant	10
12	mpi	used a wrong MPI calls	60
12	mpi	divide by zero	10
12	mpi	variable didn't initialized	30
12	mpi	missing a special case checking	100

For brevity, we give only the description of the defects: reason and effect are omitted here but this data was captured. The instrumentation did not properly capture data for this class. It is possible that there is some interaction with the system that caused the problem, as this system was used in a later study and the instrumentation also failed.

C.13 B-F05

Table C.20 B-F05 effort: sharks & fishes

	MPI		OpenMP		
Subject	Instrumented effort	Reported effort	Instrumented effort	Reported Effort	Model used first
2	8.9	20.4	4.9	25.0	openmp
3	22.8		11.1		openmp
6	23.2	16.3	1.4		mpi
7	30.8		3.1		mpi
8	21.9	19.5	3.7		mpi
9	21.4		3.4		mpi
12	14.3		0.2		mpi
14	24.3		3.4		mpi
15	14.1	14.2	16.2	13.0	openmp
16	0.8				openmp
18	10.4	9.0	5.7	6.3	openmp
19	9.6				mpi
21	9.9		2.8		mpi
22	0.8	21.6	3.1	20.8	openmp
23	29.5	18.9	11.8	16.1	openmp
24	14.0		2.2	18.2	openmp
25		12.1	0.7		mpi

C.14 H-F05

Table C.21 H-F05 effort: molecular dynamics

Subject	Model	Instrumented effort	Reported effort
1	hybrid mpi-openmp	6.8	7.5
4	hybrid mpi-openmp	2.3	
7	hybrid mpi-openmp	4.1	
8	hybrid mpi-openmp	8.7	
10	hybrid mpi-openmp	6.5	
12	hybrid mpi-openmp	6.0	
13	hybrid mpi-openmp		21.2
15	hybrid mpi-openmp	14.2	8.3
16	hybrid mpi-openmp	5.3	

Appendix D Current list of collected folklore

This list contains the current set of folklore generated by the research. We denote these F.*x,* where F stands for "folklore".

F.1 It's easier to get something working using a shared memory model than message passing.

F.2 Once you learn it, identifying parallelism is easy, but achieving performance is hard. For example, identifying parallel tasks in a computation tends to be a lot easier than getting the data decomposition and load balancing right for efficiency and scalability.

F.3 Usually, the first parallel implementation of a code is slower than its serial counterpart.

F.4 It's hard to create a parallel language that provides good performance across multiple platforms.

F.5 Debugging race conditions in shared memory programs is harder than debugging race conditions in message passing programs

F.6 Explicit distributed memory programming results in programs that run faster than shared memory programs since programmers are forced to think about data distribution (and thus locality) issues

F.7 There exist high level languages features that can be helpful for getting both SMP performance and distributed memory performance (e.g. global-array semantics). (There exists high level language features that can be helpful for getting both SMP performance and distributed memory performance.)

F.8 Use of parallel machines is not just for more CPU power, but also for more total memory or total cache (at a given level)

F.9 In master/worker parallelism, a system with a single master has limited scalability because the master becomes a bottleneck.

F.10 In MPI programs, overlapping computation and communication (non-blocking) can result in at most a 2x speedup in a program.

F.11 Sometimes, a good approach for developing parallel programs is to program for performance before programming for correctness. (Write statements that declare data structures and involve movements of data first, and wouldn't necessarily do operations on those data. Get data movement infrastructure optimized first, and then start adding mathematics.)

F.12 Given a choice, it's better to write a program with fewer large messages than many small messages. (MPI programs don't run well when you use lots of small messages because you get latency-limited)

F.13 OpenMP cannot scale beyond a small number of processors (~10)

F.14 Better programmers and better code development methodologies and tools will affect productivity much more than the next parallel programming languages.

Appendix E Folklore survey

HPC Tribal Lore Survey (A)
By Development Time working group
January 2005

For each of the HPC folklore hypotheses provided below, please check "Agree", "Disagree" or "Don't know" and explain your point of view in the comments. At the end, please write down any other HPC common beliefs that you are aware of, but are not on this list.
Before starting please complete the following sentence by checking one of the choices:

I describe myself as an expert (   ) or novice (   ) in programming high end computers.

1. It's easier to get something working using a message passing model than shared memory.
   ☐ Agree ☐ Disagree ☐ Don't know

Comments:____
____
____

2. Once you learn it, identifying parallelism is easy, but achieving performance is hard.
   ☐ Agree ☐ Disagree ☐ Don't know

Comments:____
____
____

3. It is easy to write fast code on fast machines. Generally, the first parallel implementation of code is faster than its serial counterpart.
   ☐ Agree ☐ Disagree ☐ Don't know

Comments:____
____
____

4. It's easy to create a parallel language that provides good performance across multiple platforms.
   ☐ Agree ☐ Disagree ☐ Don't know

Comments:____
____
____

5. Debugging race conditions in shared memory programs is harder than debugging race conditions in message passing programs.

☐ Agree ☐ Disagree ☐ Don’t know

Comments:____

6. Shared memory programming results in programs that run faster than explicit distributed memory programs since programmers are forced to think about data distribution (and thus locality) issues.

☐ Agree ☐ Disagree ☐ Don’t know

Comments:____

7. For large-scale shared memory systems, you can achieve better performance using OpenMP than using global arrays with explicit distribution operations.

☐ Agree ☐ Disagree ☐ Don’t know

Comments:____

8. The only reason for using parallel machines is for more CPU power.

☐ Agree ☐ Disagree ☐ Don’t know

Comments:____

9. In master/worker parallelism, a system with a single master has limited scalability because the master becomes a bottleneck.

☐ Agree ☐ Disagree ☐ Don’t know

Comments:____

10. In MPI programs, overlapping computation and communication (non-blocking) can result in at most a 2x speedup in a program.

☐ Agree ☐ Disagree ☐ Don’t know

Comments:____

11. One should always program for correctness before performance.

☐ Agree ☐ Disagree ☐ Don’t know

Comments:____

12. Given a choice, it's better to write a program with fewer large messages than many small messages.

☐ Agree ☐ Disagree ☐ Don’t know

Comments:____

Contribute your own lore

Please add your own elements of tribal lore in the space below:

HPC Tribal Lore Survey (B)

By Development Time working group

January 2005

For each of the HPC folklore hypotheses provided below, please check “Agree”, “Disagree” or “Don’t know” and explain your point of view in the comments. At the end, please write down any other HPC common beliefs that you are aware of, but are not on this list.
Before starting please complete the following sentence by checking one of the choices:

I describe myself as an expert (   ) or novice (   ) in programming high end computers.

1. It's easier to get something working using a shared memory model than message passing.

☐ Agree ☐ Disagree ☐ Don’t know

Comments:____

2. Identifying parallelism is hard, but achieving performance is easy.

☐ Agree ☐ Disagree ☐ Don’t know

Comments:____

3. It's easy to write slow code on fast machines. Generally, the first parallel implementation of a code is slower than its serial counterpart.

☐ Agree ☐ Disagree ☐ Don’t know

Comments:____

4. It's hard to create a parallel language that provides good performance across multiple platforms.

☐ Agree ☐ Disagree ☐ Don’t know

Comments:____

5. Debugging race conditions in shared memory programs is easier than debugging race conditions in message passing programs.

☐ Agree ☐ Disagree ☐ Don't know

Comments:____

6. Explicit distributed memory programming results in programs that run faster than shared memory programs since programmers are forced to think about data distribution (and thus locality) issues.

☐ Agree ☐ Disagree ☐ Don't know

Comments:____

7. For large-scale shared memory systems, you can achieve better performance using global arrays with explicit distribution operations than using Open MP.

☐ Agree ☐ Disagree ☐ Don't know

Comments:____

8. Many people use parallel machines primarily for the large amount of memory available (cache or main).

☐ Agree ☐ Disagree ☐ Don't know

Comments:____

9. In master/worker parallelism, a system with a single master will scale arbitrarily.

☐ Agree ☐ Disagree ☐ Don't know

Comments:____

10. In MPI programs, overlapping computation and communication (non-blocking) can result in an arbitrary speedup in a program.

☐ Agree ☐ Disagree ☐ Don't know

Comments:____

11. Sometimes, a good approach for developing parallel programs is to program for performance before programming for correctness.

☐ Agree ☐ Disagree ☐ Don’t know

Comments:____

12. Given a choice, it's better to write a program with many small messages than fewer large messages.

☐ Agree ☐ Disagree ☐ Don’t know

Comments:____

Contribute your own lore

Please add your own elements of tribal lore in the space below:

Appendix F Folklore survey results

	agree	disagree	don't know	blank
1	13	2	0	3
1*	2	4	3	1
1 both	17	4	3	4
2	4	5	0	1
2*	1	14	0	3
2 both	18	6	0	4
3	6	7	3	2
3*	1	8	0	1
3 both	14	8	3	3
4	9	4	2	3
4*	1	8	0	1
4 both	17	5	2	4
5	4	2	4	0
5*	4	4	5	5
5 both	8	6	9	5
6	8	6	0	4
6*	2	7	0	1
6 both	15	8	0	5
7	9	1	6	2
7*	1	5	3	1
7 both	14	2	9	3
8	5	7	4	2
8*	1	8	0	1
8 both	13	8	4	3
9	5	4	0	1
9*	0	12	4	2
9 both	17	4	4	3
10	3	4	2	1
10*	0	12	4	2
10 both	15	4	6	3
11	2	10	4	2
11*	6	3	0	1
11 both	5	16	4	3
12	4	2	3	1
12*	1	10	4	3
12 both	14	3	7	4

* indicates logical inverse of question.

Appendix G Folklore interview guide

Interview Guide

Expert Interviews

HPCS Meeting – June 27-29, 2005

Background

1. What is your formal education? Degree and Major?

2. What job functions have you performed over your career? (e.g. Developer, Manager, Scientist, …)

3. What is your current primary job function?

4. How much industrial experience do you have programming? (parallel and traditional)

5. What types of projects have you worked on in the past? (e.g. Simulation of physical phenomena, Optimization/search, Discrete event simulation, Signal/image processing , etc.)

6. What programming models are you familiar with? (Message Passing, Shared Memory, Other) How much experience do you have in each one?

7. What programming languages and platforms are you most familiar with?

Workflow

8. How do you approach a problem? On what kinds of architecture? For what kinds of product types? Using what kinds of languages?

9. How do you debug?

10. Which is more difficult - identifying parallelism or achieving a performance gain once that parallelism has been identified? Under what conditions?

11. What is the best approach for developing parallel code:
    a. Always better to get a correct solution before improving performance
    b. Better to program for performance first
    c. Something else

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

Shared Memory vs. Distributed Memory Systems

13. What are the trade-offs between a distributed message passing memory model and a shared memory model?
    a. Which model makes it easier to get code working in parallel and under what condition?
    b. Which model makes race conditions easier to debug and under what conditions?
    c. Which model produces code that run faster and under what conditions?
    d. Which model is more difficult to write a correct program and under what conditions?
    e. Does the approach to code development encouraged by either model tend to result in faster code? (e.g. distributed memory forces programmers to think about data distribution)
    f. In large shared-memory systems which is better for performance:
        i. Global arrays with explicit distribution
        ii. OpenMP

MPI

14. In an MPI program, what effect does overlapping computation and communication (non-blocking) have on the potential for speedup?

15. In message passing, what are the tradeoffs between these two approaches? Which is better?
    a. Fewer large messages
    b. More small messages

16. Do you want to tell us more about your experience with MPI?

OpenMP

17. How well can OpenMP scale for large number of processors?

18. Do you want to tell us more about your experience with OpenMP?

New Languages

19. What do you think about Co-Array Fortran?
20. What do you think about UPC?

Lessons Learned

For the next questions think about the most difficult code you have worked on (or your latest experience):

21. Briefly describe the experience:
    Problem solved, Language used, Programming model used, Platform used

22. What made it difficult (or what was your main difficulty)?

23. How did you (or your organization) decide what technologies (language, programming model) to use for solving the problem? What were the main factors that affected this decision?

24. If you were to do it again what would you do the same or different in terms of:
    a. Programming language
    b. Programming model
    c. Platform
    d. Other

Wrap-up questions

25. Looking back at your experience, what are the strongest beliefs that you have about high-end computing (general beliefs, programming models features, language features, platform features, ….)

26. Looking at the entire community, what do you think are the main expectations from new technologies? (features that are missing in the existing technologies)

27. Is there anything that you would like to add?

28. Did we miss anything? Do you have questions for us?

Appendix H ASC-Alliance 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 chem eng, 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 I ASC-Alliance interview guide

Product

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)?

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?

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?

Project organization

Organizational structure

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

Staff

How many of the project members are core developers?

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?

Using the software

Users

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

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?

Examining the output

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

Development activities

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 codebase?

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?

Testing

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

How do you evaluate new algorithms?

Tuning

When do you tuning?

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?

Debugging

Which strategies do you use for debugging?

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

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?

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

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)

Programming models and productivity

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?

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 for it to be worth it for you to adopt it?

Productivity measures

What is a meaningful unit of "productivity" to you?

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 you development effort/ achieved performance? Would you be willing to have a researcher do an observational study on some programmers?

Appendix J Classroom case study interview guide

General problem questions

1. Can you briefly go over the problem you were trying to solve?

2. Were you trying to solve a problem in less time, or trying to solve a larger problem?

3. How did you exploit parallelism in this problem?

4. If implemented in distributed memory model, how did you do the domain decomposition?

5. Were you starting from scratch, or from some existing code?

Organization

6. How did you split up the work among you?

7. Did you have any problems coordinating efforts?

8. If not everyone wrote parallel code, how did you decide who would it?

Process

9. Did you have a fully working version of the serial code before you started parallelizing?

Effort

10. Percentage-wise, how did the effort break down in terms of

- planning
- serial coding
- parallel coding
- tuning / optimizing

1. Did you finish on schedule, or did it take you longer than you thought it would? If so, why? Was this related to the programming model?

Effort – parallelizing

12. How much of the parallelizing effort was due to the programming model, and how much was due to other issues (e.g. remote development issues, compiler maturity issues, build issues)?

13. If wrote serial from scratch, what took more time, writing the serial, or doing the parallel?

14. If ported serial from other language, what took more time, porting the serial, or doing the parallelizing?

Effort – testing/debugging

15. How did you test your code?

16. Roughly speaking, how much of your total effort (percentage/hours) was spent testing and debugging?

17. Can you break this down into serial/parallel?

18. Did you encounter any particularly difficult bugs? Can you describe them? Were they "serial" or "parallel" related?

18a. Did you have any integration problems?

19. Did you use any debugging tools? If not, why not?

Effort – tuning

20. Did you get a performance improvement on your first correct parallel solution?

21. Did you need to do any tuning?

22. If so, did you tune only at the last stage of your development, or did you begin tuning before you had implemented all of the features of your system?

23. Did you use any profiling tools? If not, why not?

Effort – other

24. Did you encounter other tasks we didn't mention that took up a significant fraction of your time?

25. Did you have trouble / spend a lot of time getting the program to compile and run (e.g. makefiles, run scripts)?

Performance

26. What was the program performance like?

27. Did you get a significant performance boost when running in parallel?
(much better than serial?)

Comparing to other kinds of development

28. What aspects of the process were harder relative to the other type of software development you've done?

29. What did you miss from other tools/languages?

Comparing to class studies

30. What were some of the differences between working on this project and solving the classroom assignment?

30a. Did you feel that you were more proficient in the programming model after having done the assignment?

Retrospective

31. Would you consider the project a "success" or a "failure" in terms of trying to achieve your goals?

32. In retrospect, do you think this was a suitable problem to try and parallelize? (e.g. would benefit from HPC and did not require too much effort?)

33. If you had to do over again, would you have chosen a different model?

34. Did you look into using any parallel libraries as an alternative to coding in the base model?

35. Tools?

36. Anything else you would like to add?

Appendix K Assignment template

Note: This template was mostly developed by Andy Funk at MIT Lincoln Labs.

Assignment Template

HEADER

Course:
Title:
Date Assigned:
Date Due:

ASSIGNMENT GOAL

Briefly describe what student is supposed to learn about parallel programming.

PROBLEM STATEMENT

Specify as completely as possible the requirements for the code to be implemented.

SPECIFIC INSTRUCTIONS

Specify which languages(s) and memory model(s) are to be used.
Specify whether or not a serial version of the same problem should be used as a starting point for the parallel version. If there will be a serial version, will it be given or must it be developed by the student?
If the students should create multiple versions of the assignment, using multiple languages and/or memory models, you may also specify here the due date for each version.

INPUT

Describe the input values and parameters on which the student must run the code.
For example, for grid of resistors or game of life, specify which grid sizes must be demonstrated.
If there are files that the students should download containing specific input values, describe where they can be found.

OUTPUT

Specify what results should be returned. Give the students a matrix in which to fill out execution times for their implementation, based on the following:

Size of Input Dataset / Processors	*A*	*B*	*C*...
X			
Y			
Z			

HINTS

Provide any programming guidelines that can help students better achieve a solution.

PLATFORM

Record information about the machine on which the assignments will be developed and run.

COMPILER

Instructions for using the compiler and any other tools.

HOW TO SUBMIT

Give detailed instructions as to how to submit (possibly multiple versions of) the source code and program output, as well as the development and runtime measures (as specified in the "output" section).
Detail any file naming conventions that must be followed, or other instructions.

GRADING

Provide as clearly as possible the grading criteria.

ADDITIONAL RESOURCES

List any shared source code, web links, etc.

Glossary

AHPCRC: Army High Performance Computing Research Center

ARSC: Arctic Research Supercomputing Center

CAF: Co-Array Fortran, a PGAS language which extends Fortran by adding co-arrays.

codes: A term commonly used by HPC practitioners to refer to programs.

DARPA: Defense Advanced Research Projects Agency. The research and development agency for the United States Department of Defense.

DoD: Department of Defense

DOE: Department of Energy

HEC: High-end computing

HPC: High-performance computing. A synonym of high-end computing.

HPCS: High Productivity Computing Systems. A project sponsored by DARPA to develop the next generation of HEC systems.

HPF: High-Performance Fortran. A data-parallel version of Fortran that did not achieve widespread adoption in the United States.

MPI: Message-passing interface. Commonly used message-passing library for parallel programming.

OpenMP: A parallel, multi-threaded language extension for C and Fortran.

PGAS: Partitioned-global address space. PGAS languages include UPC, Co-Array Fortran, and Titanium

SDSC: San Diego Supercomputing Center

SMP: Symmetric multi-processor. On an SMP machine, the main memory is physically shared across all of the processors. Such systems are sometimes referred to as “shared memory” systems, in contrast with “distributed memory” systems.

UPC: Unified-Parallel C. A PGAS language which extends the C language by adding parallel constructs.

Bibliography

[Armstrong99]	Armstrong, R.; Gannon, D.; Geist, A.; Keahey, K.; Kohn, S.; McInnes, L.; Parker, S. & Smolinski, B. (1999),Toward a Common Component Architecture for High-Performance Scientific Computing, *in* 'Proceedings of the 1999 Conference on High Performance Distributed Computing'.
[Bailey81]	Bailey., J. & Basili, V.R. (1981), A Meta-Model for Software Development Resource Expenditures, *in* 'Proceedings of the Fifth International Conference on Software Engineering', pp. 107-116.
[Baley91]	Bailey, D.H. (1991), 'Twelve Ways to Fool the Masses When Giving Performance Results on Parallel Computers', *Supercomputing Review* **4**(8), pp. 54-55.
[Basili89]	Basili, V.R. (1989),Software Development: A Paradigm for the Future, *in* 'Proceedings of the 13th International Computer Software and Applications Conference (COMPSAC 89)', pp. 471-485.
[Basili94]	Basili, V.R.; Caldiera, G. & Rombach, H.D. (1994),The Experience Factory *in* 'Encyclopedia of Software Engineering', John Wiley & Sons, Inc., , pp. 469-476.
[Basili01]	Basili, V.R.; Tesoriero, R.; Costa, P.; Lindvall, M.; Rus, I.; Shull, F. & Zelkowitz, M. (2001),Building an Experience Base for Software Engineering: A report on the first CeBASE eWorkshop, *in* 'Proceedings of the Product Focused Software Process Improvement Conference'.
[Basili06]	Basili, V.R. & Elbaum, S. (2006),'Better Empirical Science for Software Engineering', 28th International Conference on Software Engineering (ICSE 2006), invited talk.
[Berthou01]	Berthou, J.Y. & Fayolle, E. (2001), 'Comparing OpenMP, HPF, and MPI Programming: A Study Case', *The International Journal of High Performance Computing Applications* **15**(3), pp. 297-309.
[Boehm81]	Boehm, B.W. (1981), *Software Engineering Economics*, Prentice Hall PTR.

[Bokus94]	Bokus, Z.; Choudhary, A.; Fox, G.; Haupt, T.; Ranka, S. & Wu, M.Y. (1994), 'Compiling Fortran 90D/HPF for Distributed Memory MIMD Computers', *Journal of Parallel and Distributed Computing* **21**(1), pp.15-26.
[Brooks97]	Brooks, A. (1997), 'Meta Analysis-A Silver Bullet-for Meta-Analysts', *Empirical Software Engineering* **2**(4), pp. 333-338.
[Campbell63]	Campbell, D.T. & Stanley, J.C. (1963), *Experimental and Quasi-Experimental Designs for Research*, Houghton Mifflin, Boston.
[Cantonnet05]	Cantonnet, F.; Yao, Y.; Zahran, M. & El-Ghazawi, T. (2004),Productivity Analysis of the UPC Language, *in* 'Proceedings of the IPDPS 2004 PMEO workshop'.
[Carlson99]	Carlson, W.; Culler, D.; Yelick, K.; Brooks, E. & Warren, K. (1999),'Introduction to UPC and Language Specification' (CCS-TR-99-157), Technical report, Center for Computing Sciences.
[Chamberlain98]	Chamberlain, B.L.; Sung-Eun, C.; Lewis, E.C.; Snyder, L. & Weathersby, W.D. (1998), 'The Case for High-Level Parallel Programming in ZPL', *IEEE Computational Science & Engineering* **5**(3), pp. 76-86.
[Chamberlain00]	Chamberlain, B.L.; Dietz, S. & Snyder, L. (2000), A Comparative Study of the NAS MG Benchmark Across Parallel Languages and Architectures, *in* 'Proceedings of the 2000 ACM/IEEE Supercomputing Conference on High Performance Networking and Computing (SC2000)', pp. 297-310.
[Chandra01]	Chandra, R.; Dagum, L.; Kohr, D.; Maydan, D.; McDonald, J. & Menon, R. (2000), *Parallel Programming in OpenMP*, Morgan Kaufman
[Choy03]	Choy, R. & Edelman, A. (2003),MATLAB*P 2.0: A Unified Parallel MATLAB, *in* 'Singapore MIT Alliance Symposium'.
[Cohen94]	Cohen, J. (1994), 'The Earth Is Round (p<.05)', *American Psychologist* **49**(12), pp. 997-1003.

[Cook97] Cook, D.; Mulrow, C. & Haynes, R. (1997), 'Systematic Reviews: Synthesis of Best Evidence for Clinical Decisions', *Annals of Internal Medicine* **126**(5), pp. 376-380.

[Dagum98] Dagum, L. & Menon, R. (1998), 'OpenMP: An Industry-Standard API for Shared-Memory Programming', *IEEE Computational Science & Engineering* **5**(1), pp. 46-55.

[Data05] Datta, K.; Bonachea, D. & Yelick, K. (2005), Titanium Performance and Potential: an NPB Experimental Study, *in* 'Proceedings of the Workshop on Languages and Compilers for Parallel Computing (LCPC)'.

[Dongarra96] Dongarra, J.J.; Otto, S.W.; Snir, M. & Walker, D. (1996), 'A message passing standard for MPP and workstations', *Communications of the ACM* **39**(7), pp. 84-90.

[Dongarra03] Dongarra, J.J.; Foster, I.; Fox, G.; Kennedy, K.; Torczon, L. & White, A. (2003), *Sourcebook of Parallel Computing*.

[Dyba05] Dyba, T.; Kitchenham, B.A. & Jorgensen, M. (2005), 'Evidence-based Software Engineering for Practitioners', *IEEE Software* **22**(1), pp. 58-65.

[Faulk04] Faulk, S.; Gustafson, J.; Johnson, P.; Porter, A.; Tichy, W. & Votta., L. (2004), 'Measuring HPC Productivity', *The International Journal of High Performance Computing Applications* **18**(4).

[Fenton93] Fenton, N. (1993), 'How Effective are Software Engineering Methods?', *Journal of Systems and Software* **22**(2), pp. 141-146.

[Fox90] Fox, G.C.; Hirandani, S.; Kennedy, K.; Koelbel, C.; Kremer, U.; Tseng, C. & Wu, M. (1990),'Fortran D Language Specification' (CRPC-TR 90079), Technical report, Rice and Syracuse University.

[Glass94] Glass, R. (1994), 'The Software Research Crisis', *IEEE Software* **11**(6), 42-46.

[Glass02] Glass, R.; Vessey, I. & Ramesh, V. (2002), 'Research in Software Engineering: an Analysis of the Literature', *Information and Software Technology* **44**, 491-506.

[Glass03] Glass, R. (2003), 'The State of the Practice of Software Engineering', *IEEE Software* **20**(6), 20-21.

[Graham04] Graham, S.; Snir, M. & Patterson, C.A. (2004),'Getting Up To Speed: The Future of Supercomputing', Technical report, National Research Council.

[Gropp94] Gropp, W.; Lusk, E. & Skjellum, A. (1994), *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, MIT Press.

[Gurd85] Gurd, J.R.; Kirkham, C.C. & Watson, I. (1985), 'The Manchester prototype dataflow computer', *Communications of the ACM* **28**(1), pp. 34-52.

[HECRTF04] Federal Plan for High-End Computing: Report of the High-End Computing Revitalization Task Force (HECRTF), (2004).

[Hochstein05] Hochstein, L.; Basili, V.R.; Zelkowitz, M.; Hollingsworth, J. & Carver, J. (2005), Combining self-reported and automatic data to improve programming effort measurement, *in* 'Fifth joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE'05)'.

[Host05] Host, M.; Wohlin, C. & Thelin, T. (2005), Empirical Context Classification: Incentives and Experience of Subjects, *in* 'Proceedings of the 27th international conference on Software engineering'.

[IEEE-610.12] IEEE Standard Glossary of Software Engineering Terminology'(IEEE Std 610.12-1990).

[IEEE-1045] IEEE Standard for Software Productivity Metrics (IEEE Std 1045-1992),

[JáJá92] JáJá, J. (1992), *An Introduction to Parallel Algorithms*, Addisohn-Wesley Professional.

[Johnson97] Johnson, E. & Gannon, D. (1997),HPC++: Experiments with the Parallel Standard Template Library, *in* 'Proceedings of the 11th International Conference on Supercomputing', pp. 124-131.

[Johnson03] Johnson, P.M.; Kou, H.; Agustin, J.; Chan, C.; Moore, C.; Miglani, J.; Zhen, S. & Doane, W.E. (2003), Beyond the Personal Software Process: Metrics collection and analysis for the differently disciplined, *in* 'Proceedings of the 25th International Conference on Software Engineering.'

[Joseph04] Joseph, E.; Snell., A. & Willard, C.G. (2004),'Council on Competitiveness Study of U.S. Industrial HPC Users', Technical report, Council on Competitiveness.

[Kale93] Kale, L.V. & Krishnan, S. (1993), CHARM++ : A Portable Concurrent Object Oriented System Based On C++, *in* 'Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications', pp. 91-108.

[Kennedy04] Kennedy, K.; Koelbel, C. & Schreiber, R. (2004), 'Defining and Measuring the Productivity of Programming Languages', *The International Journal of High Performance Computing Applications* **18**(4).

[Kepner04] Kepner, J. (2004), 'HPC Productivity: An Overarching View', *The International Journal of High Performance Computing Applications* **18**(4), pp. 393-397.

[Kitchenham02] Kitchenham, B.A.; Pfleeger, S.L.; Pickard, L.M.; Jones, P.W.; Hoaglin, D.C.; El Emam, K. & Rosenberg, J. (2002), 'Preliminary Guidelines for Empirical Research in Software Engineering', *IEEE Transactions on Software Engineering* **28**(8).

[Krosnick99] Krosnick, J. (1999), 'Survey Research', *Annual Review of Psychology* **50**, pp. 537-567.

[Kuck04] Kuck, D.J. (2004), 'Productivity in High Performance Computing', *The International Journal of High Performance Computing Applications*.

[Lethbridge05] Lethbridge, T.; Sim, S.E. & Singer, J. (2005), 'Studying Software Engineers: Data Collection Techniques for Software Field Studies', *Empirical Software Engineering* **10**(3), pp. 311-341.

[Miller99] Miller, J. (1999), Can Results from Software Engineering Experiments be Safely Combined?, *in* 'Proceedings of the Sixth International Software Metrics Symposim', pp. 152-158.

[Miller00]	Miller, J. (2000), 'Applying Meta-analytical Procedures to Software Engineering Experiments', *Journal of Systems and Software* **54**(1), pp. 29-39.
[Morton95]	Morton, D.; Wang, K. & Ogbe, D.O. (1995), 'Lessons Learned in Porting Fortran/PVM Code to the Cray T3D', *IEEE Parallel & Distributed Technology: Systems & Applications* **3**(1), pp. 4-11.
[Nakamura06]	Nakamura, T.; Hochstein, L. & Basili, V.R. (2006), Identifying Domain-Specific Defect Classes Using Inspections and Change History, *in* 'Proceedings of the 5th International Symposium on Empirical Software Engineering (ISESE'06)'.
[Narlikar98]	Narlikar, G.J. & Blelloch, G.E. (1998),Pthreads for Dynamic and Irregular Parallelism, *in* 'Proceedings of the 1998 ACM/IEEE conference on Supercomputing'.
[NAS-95-020]	Bailey, D.; Harris, T.; Saphir, W.; van der Wijngaart, R.; Woo, A. & Yarrow, M. (1995),'The NAS Parallel Benchmarks 2.0' (NAS-95-020), Technical report, NASA.
[Nieplocha96]	Nieplocha, J.; Harrison, R.J. & Littlefield, R.J. (1996), 'Global Arrays: A Nonuniform Memory Access Programming Model for High-Performance Computers', *Journal of Supercomputing* **10**(2), pp. 169-189.
[NSF05]	'2005 NSF Cyberinfrastructure User Survey', Technical report, National Science Foundation.
[Numrich98]	Numrich, R.W. & Reid, J.K. (1998), 'Co-Array Fortran for Parallel Programming'.
[Oldham02]	Oldham, J.D. (2002), 'Scientific Computing Using POOMA', *C/C++ Users Journal* **20**(6).
[Pancake94]	Pancake, C.M. & Cook, C. (1994),What Users Need in Parallel Tool Support: Survey Results and Analysis, *in* 'Proceedings of the Scalable High-Performance Computing Conference'.
[Pancake96]	Pancake, C.M. (1996), 'Is Parallelism for You?', *IEEE Computational Science & Engineering* **3**(2), pp. 18-37.
[Pancake97]	Pancake, C.M. (1997), 'Establishing Standards for HPC System Software and Tools', *NHSE Review* **1997**(1).

[Pancake00] Pancake, C.M. & Bergmark, D. (2000), 'Do Parallel Languages Respond to the Needs of Scientific Programmers?', *IEEE Computer* **23**(12), pp. 13-23.

[Perry96] Perry, D.E.; Staudenmayer, N.A. & Votta, L.G. (1996), *Understanding and Improving Time Usage in Software Development*, John Wiley and Sons, chapter 5.

[Perry04] Perry, D.E.; Sim, S.E. & Easterbrook, S.M. (2004), Case Studies for Software Engineers, *in* 'Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)', pp. 736-738.

[Philippsen97] Philippsen, M. & Zenger, M. (1997), 'JavaParty - Transparent Remote Objects in Java', *Concurrency: Practice and Experience* **9**(11), pp. 1225-1242.

[Pickard98] Pickard, L.M.; Kitchenham, B.A. & Jones, P.W. (1998), 'Combining Empirical Results in Software Engineering', *Information and Software Technology* **40**(14), 811-821.

[PITAC05] Benioff, M.R. & Lazowska, E.D. (2005), 'Computational Science: Ensuring America's Competitiveness', President's Information Technology Advisory Committee (PITAC).

[Post03] Post, D. & Kendall, R. (2003), 'Software Project Management and Quality Engineering Practices for Complex, Coupled Multi-Physics, Massively Parallel Computational Simulations: Lessons Learned from ASCI', *The International Journal of High Performance Computing Applications* **18**(4).

[Robson02] Robson, C. (2002), *Real World Research*, Blackwell Publishing.

[Rodman99] Rodman, A. & Brorsson, M. (1999), Programming Effort vs. Performance with a Hybrid Programming Model for Distributed Memory Parallel Architectures, *in* 'Euro-Par'99 parallel processing: 5th International Euro-Par Conference', pp. 888-898.

[Sahni96] Sahni, S. & Thanvantri, V. (1996), 'Performance Metrics: Keeping the Focus on Runtime', *IEEE Parallel & Distributed Technology* **4**(1), 43-56.

[Schmidt92]	Schmidt, F.L. (1992), 'What do Data Really Mean? Research Findings, Meta-analysis and Cumulative Knowledge in Psychology', *American Psychologist* **47**(10), pp. 1173-1181.
[Seaman95]	Seaman, C. (1995), 'Qualitative Methods in Empirical Studies of Software Engineering', *IEEE Transactions on Software Engineering* **25**(4), pp. 557-572.
[Shull05]	Shull, F.; Basili, V.R.; Carver, J. & Hochstein, L. (2005), Empirical study design in the area of High-Performance Computing (HPC), *in* 'Proceedings of the 4th International Symposium on Empirical Software Engineering'.
[Sjoberg05]	Sjoberg, D.; Hannay, J.; Hansen, O.; Kampenes, V.; Karahasanovic, A.; Liborg, N. & Rekdal, A.C. (2005), 'A Survey of Controlled Experiments in Software Engineering', *IEEE Transactions on Software Engineering* **31**(9), 733-753.
[Snir04]	Snir, M. & Bader, D.A. (2004), 'A Framework for Measuring Supercomputer Productivity', *The International Journal of High Performance Computing Applications* **18**(4), 417-432.
[Sterling04]	Sterling, T. (2004), 'Productivity Metrics and Models for High Performance Computing', *The International Journal of High Performance Computing Applications* **18**(4).
[Strauss05]	Strauss, S.E.; Richardson, W.S.; Glasziou, P. & Haynes, R.B. (2005), *Evidence-Based Medicine: How to Practice and Teach EBM*, Churchill Livingstone.
[Sunderman94]	Sunderam, V.S.; Geist, G.A.; Dongarra, J.J. & Manchek, R. (1994), 'The PVM Concurrent Computing System: Evolution, Experiences, and Trends', *Parallel Computing* **20**(4), 531-545.
[Szafron96]	Szafron, D. & Schaeffer, J. (1996), 'An Experiment to Measure the Usability of Parallel Programming Systems', *Concurrency: Practice and Experience* **8**(2), 147-166.
[Vegas02]	Vegas, S. (2002), 'Characterization Schema for Selecting Software Testing Techniques', PhD Thesis, Universidad Politecnica de Madrid.
[Vishkin98]	Vishkin, U.; Dascal, S.; Berkovich, E. & Nuzman, J. (1998), Explicit Multi-threading (XMT) Bridging Models for Instruction Parallelism, *in* 'ACM Symposium on Parallel Algorithms and Architectures'.

[Walston77] Walston, C. & Felix, C. (1977), 'A Method of Programming Measurement and Estimation', *IBM Systems Journal* **16**(1), pp. 84-95.

[Wohlin04] Wohlin, C. (2004), 'Are Individual Differences in Software Development Performance Possible to Capture Using a Quantitative Survey?', *Empirical Software Engineering* **9**(3), pp. 211-228.

[Yelick98] Yelick, K.A.; Semenzato, J.; Pike, G.; Miyamoto, C.; Liblit, B.; Krishnamurthy, A.; Hlifinger, P.N.; Graham, S.L.; Gay, D.; Colella, P. & Aiken, A. (1998), 'Titanium: A High-Performance Java Dialect', *Concurrency: Practice and Experience* **10**(11-13).

[Yin94] Yin, R.K. (1994), *Case Study Research: Design and Methods*, SAGE Publications.

[Zelkowitz05] Zelkowitz, M.; Basili, V.R.; Asgari, S.; Hochstein, L.; Hollingsworth, J. & Nakamura, T. (2005), Measuring Productivity on High Performance Computers, *in* 'Proceedings of the 11th International Symposium on Software Metrics'.