Experiments to Understand HPC Time to Development

(Final report for Department of Energy contract DE-FG02-04ER25633)
Report DOE/ER/25633-1

Victor Basili[1,2]
Marvin Zelkowitz[1]

with contributions by
Lorin Hochstein[4], Taiga Nakamura[6], Sima Asgari[1],
Jeffrey K. Hollingsworth[1], Forrest Shull[2], Rola Alameh[1]
Jeffrey Carver[3], Martin Voelp[1], Nico Zazworka[1], Philip Johnson [5]

[1]University of Maryland, College Park **[2]Fraunhofer Center, Maryland**
[3]Mississippi State University **[4]University of Nebraska - Lincoln**
[5]University of Hawaii **[6]IBM Japan**

November 14, 2007

Abstract

In order to understand how high performance computing (HPC) programs are developed, a series of experiments, using students in graduate level HPC classes and various research centers, were conducted at various locations in the US. In this report, we discuss this research, give some of the early results of those experiments, and describe a web-based Experiment Manager we are developing that allows us to run studies more easily and consistently at universities and laboratories, allowing us to generate results that more accurately reflect the process of building HPC programs.

Table of Contents

1 Introduction

The DARPA High Productivity Computing Systems (HPCS) project has goals of "providing a new generation of economically viable high productivity computing systems for national security and for the industrial user community," and initiating "a fundamental reassessment of how we define and measure performance, programmability, portability, robustness and ultimately, productivity in the HPC domain"[1].

In order to reassess the definitions and measures in a scientific domain it is necessary to study the basis and source of those definitions and measures. These sources are usually found in the related literature and various documentations existent in the community. However the large amount of tacit information that is merely in people's minds often remains neglected.

Historically, there has been little interaction between the HPC and the software engineering communities. The Development Time Working Group of the HPCS project is focused on development time issues. The group has both software engineering researchers as well as HPC researchers. The strategy of the working group is to apply empirical methods to study parallel programming issues. We have applied similar methods in the past to researching development time issues in other software domains [BA02].

Much of the literature in the Software Engineering community concerning programmer productivity was developed with assumptions that do not necessarily hold in the High Performance Computing (HPC) community:

1. In scientific computation insights culled from results of one program version often drives the needs for the next. The software itself is helping to push the frontiers of understanding rather than the software being used to automate well-understood tasks.
2. The requirements often include conformance to sophisticated mathematical models. Indeed, requirements may often take the form of an executable model in a system such as Mathematica, and the implementation involves porting this model to HPC systems.
3. "Usability" in the context of an HPC application development may revolve around optimization to the machine architecture so that computations complete in a reasonable amount of time. The effort and resources involved in such optimization may exceed initial development of the algorithm.

Due to these unique requirements, traditional software engineering approaches for improving productivity may not be directly applicable to the HPC environment.

As a way to understand these differences, we are developing a set of tools and protocols to study programmer productivity in the HPC community. Our initial efforts have been to understand the effort involved and defects made in developing such programs. We also want to develop models of workflows that accurately explain the process that HPC programmers use to build their codes. Issues such as time involved in developing serial and parallel versions of a program, testing and debugging of the code, optimizing the code for a specific parallelization model (e.g., MPI, OpenMP) and tuning for a specific machine architecture are all topics of study. If we have those models, we can then work on the more crucial problems of what tools and techniques better optimize a programmer's performance to produce quality code more efficiently.

Since 2004 we have been conducting human-subject experiments at various locations across the U.S. in graduate level HPC courses and with interviews with professional programmer at HPC centers (Figure 1). Graduate students in a HPC class are fairly typical of a large class of novice HPC programmers who may have years of experience in their application domain but very little in HPC-style programming. Multiple students are routinely given the same assignment to perform, and we conduct experiments to control for the skills of specific programmers (e.g., experimental meta-analysis) in different environments. Due to the relatively low costs, student studies are an excellent environment to debug protocols that might be later used on practicing HPC programmers.

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

1.1 Classroom as software engineering lab

The classroom is an appealing environment for conducting software engineering experiments, for several reasons:

- Most researchers are located at universities. Being close to your subjects is often necessary to obtain accurate results.
- Training can be integrated into the course. No extra effort is then required by the subjects since there is the assumption that the training is a valuable academic addition to the classroom syllabus.
- Required tasks can be integrated into the course.
- All subjects are performing identical programming tasks, which is not generally true in industry. This provides an easy source for replicated experiments.

In addition to the results that are obtained directly by these studies, such experiments are also useful for piloting experimental designs and protocols which can later be applied to industry subjects, an approach which has been used successfully elsewhere (e.g., [BA94] [BA97] [BA99]).

While there are threats to validity of such studies by using students as subjects as proxies for professional programmers (e.g., the student environment may not be representative of the ones faced by professional programmers), there are additional complexities that are specific to research in this type of environment. We encountered each of these issues when conducting research on the effects of parallel programming model on effort in high-performance computing [HO05b]:

1. Complexity. Conducting an experiment in a classroom environment is a complex process that requires many different activities (e.g., planning the experimental design, identifying appropriate artifacts and treatments, enrolling students, providing for data collection, checking for process compliance, sanitizing data for privacy, analyzing data). Each such activity identifies multiple points of failure, thus requiring a large effort to organize and run multiple studies. If the study is done at multiple universities in collaboration with other professors, these professors may have no experience in organizing and conducting such experiments.

2. Research vs. pedagogy. When the experiment is integrated into a course, the experimentalist must take care to balance research and pedagogy [CA03]. Studies must have minimal interference with the course. If the students in one class are divided up into treatment groups and the task is part of an assignment, then care must be taken to ensure that the assignment is of equivalent difficulty across groups. Students who consent to participate must not have any advantage or disadvantage over students who do not consent to participate, which limits additional overhead required by the experiment. In fact, each university's Institutional Review Board (IRB), required in all USA universities performing experiments with human subjects, insists that participation (or non-participation) must have no effect on the student's grade in the course.

3. Consistent replication across classes. To build empirical knowledge with confidence, researchers replicate studies in different environments. If studies are to be replicated in different classes, then care must be taken to ensure that the artifacts and data collection protocols are consistent. This can be quite challenging because professors have their own style of giving assignments. Common projects across multiple locations often differ in crucial ways making meta-analysis of the combined results impossible [MI00].

4. Participation overhead for professors. In our experience, many professors are quite willing to integrate software engineering studies into their classroom environment. However, for professors who are unfamiliar with experimental protocols, the more effort required of them to conduct a study, the less likely it will be a success. In addition, collaborating professors who are not empirical researchers may not have the resources or the inclination to monitor the quality of captured data to evaluate process conformance. Therefore, empirical researchers must try to minimize any additional effort required to run an empirical study in the course while ensuring that data is being captured correctly.

The required IRB approval, when attempted for the first time, seems like a formidable task. Help in understanding IRB approval would greatly aid the ability of conducting such research experiments.

5. Participation overhead for students. An advantage of integrating a study into a classroom environment is that the students are already required to perform the assigned task as part of the course, so the additional effort involved in participating in the study is much lower than if subjects were recruited from elsewhere. However, while the additional overhead is low, it is not zero. The motivation to conform to the data collection process is, in general, much lower than the motivation to perform the task, because process conformance cannot be graded. In addition, the study should not subvert the educational goals of the course. Putting the experiment in the context of the course syllabus is never easy.

This can be particularly problematic when trying to collect process data from subjects (e.g. effort, activities, defects), especially for assignments that take several weeks. (e.g., We saw a reduction in process conformance over time when subjects had to fill out effort logs over the course of multiple assignments).

6. Automatic data collection of software process. To reduce subject overhead and increase data accuracy, it is possible to collect data automatically from the programmer's environment. Capturing data at the right level of granularity is difficult. All user-generated events can be captured (keyboard events, mouse events), but this produces an enormous volume of data that may not abstract to useful information. Allowing this raw data to be used can create privacy issues, such as revealing account names, with the ability to then determine how long specific users took to build a product or how many defects they made.

All development activities taking place within a particular development environment (e.g., Eclipse) simplifies the task of data collection, and tools exist to support such cases (e.g. Marmoset [SP05]). However, in many domains development will involve a wide range of tools and possibly even multiple machines. For example in the domain of high-performance computing, preliminary programs may be compiled on a home PC, final programs are developed on the university multiprocessor and are ultimately run on remote supercomputers at a distant datacenter. Programmers typically use a wide variety of tools, including editors, compilers, build tools, debuggers, profilers, job submission systems, and even web browsers for viewing documentation.

7. Data management. Conducting multiple studies generates an enormous volume of heterogeneous data. Along with automatically collected data and manually reported data, additional data includes versions of the programs, pre- and post-questionnaires, and various quality outcome measures (e.g. grades, code performance, defects). Because of privacy issues, and to conform to IRB regulations, all data must be stored with appropriate access controls, and any exported data must be appropriately sanitized. Managing this data manually is labor-intensive and error-prone, especially when conducting studies at multiple sites.

Limitations of student studies include the relatively short programming assignments possible due to the limited time in a semester and the fact these assignments must be picked for the educational value to the students as well as their investigative value to the research team.

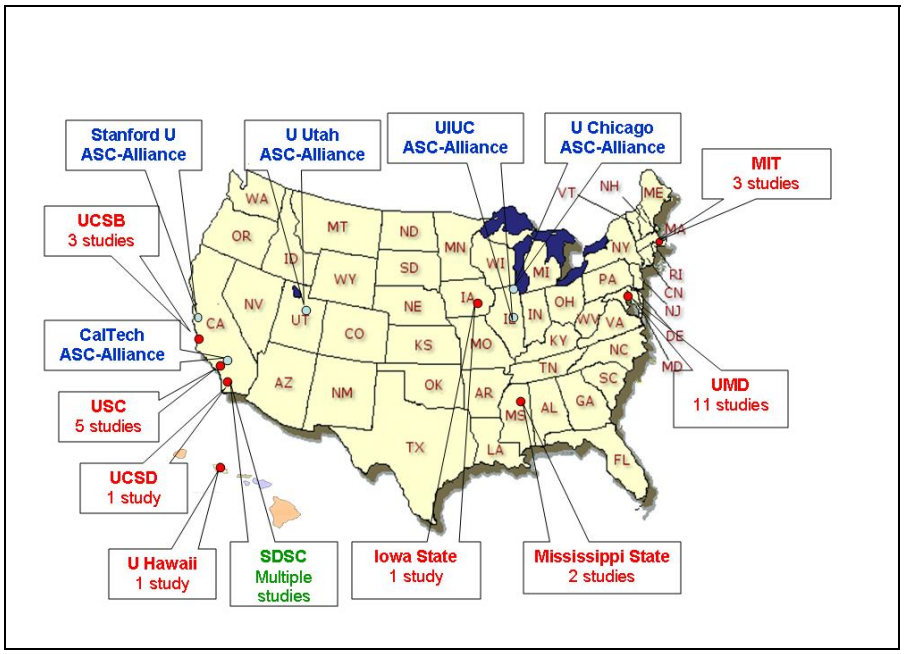

Figure 1. Studies conducted.

In this report, we present both the methodology we have developed to investigate programmer productivity issues in the HPC domain (Section 2), some initial results of studying productivity of novice HPC programmers (Section 3).

2 Experiment methodology

In each class, we obtained consent from students to be part of our study. There is a requirement at every U.S. institution that studies involving human subjects must be approved by that university's Institutional Review Board (IRB). The nature of the assignments was left to the individual instructors for each class since instructors had individual goals for their courses and the courses themselves had different syllabi. However, based on previous discussions as part of this project, many of the instructors used the same assignments (Table 1), and we have been collecting a database of project descriptions as part of our Experiment Manager website (See Section 3.1). To ensure that the data from the study would not impact students' grades (and a requirement of almost every IRB), our protocol quarantined the data collected in a class from professors and teaching assistants for that class until final grades had been assigned.

Embarrassingly parallel:
Buffon-Laplace needle problem, Dense matrix-vector multiply
Nearest neighbor:
Game of life, Sharks & fishes, Grid of resistors, Laplace's equation, Quantum dynamics
All-to-all:
Sparse matrix-vector multiply, Sparse conjugate gradient, Matrix power via prefix
Shared memory:
LU decomposition, Shallow water model, Randomized selection, Breadth-first search
Other:
Sorting

Table 1: Sample programming assignments

We need to measure the time students spend working on programming assignments with the task that they are working on at that time (e.g. serial coding, parallelization, debugging, tuning). We used three distinct methods: (1) explicit recording by subject in diaries (either paper or web-based); (2) implicit recording by instrumenting the development environment; and (3) sampling by an operating system installed tool (e.g., Hackystat [Jo04]). Each of these approaches has strengths and limitations. But significantly, they all give different answers. After conducting a series of tests using variations on these

techniques, we settled on a hybrid approach that combines diaries with an instrumented programming environment that captures a time-stamped record of all compiler invocations (including capture of source code), all programs invoked by the subject as a shell command, and interactions with supported editors. Elsewhere [Ho05], we describe the details of how we gather this information and convert it into a record of programmer effort.

After students completed an assignment, the data was transmitted to the University of Maryland, where it was added to our Experiment Manager database. Looking at the database allows post-project analysis to be conducted to study the various hypotheses we have collected via our folklore collection process.

For example, given workflow data from a set of students, the following hypotheses that are the subjective opinion of many in the HPCS community, collected via surveys at several HPCS meetings, can be tested [As05]:

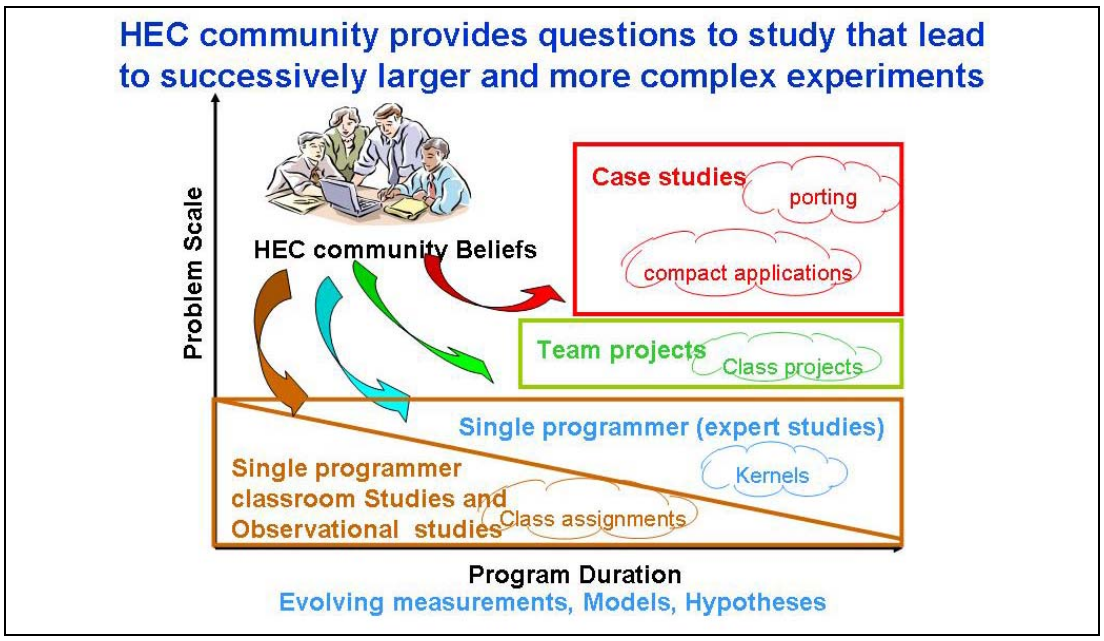

Figure 2. Research Plan.

Hyp 1: *The average time to fix a defect due to race conditions will be longer in a shared memory program compared to a message-passing program*. To test this hypothesis we can measure the time to fix defects due to race conditions.

Hyp. 2: *On average, shared memory programs will require less effort than message passing, but the shared memory outliers will be greater than the message passing outliers.* To test this hypothesis we measure the total development time.

Hyp. 3: *There will be more students who submit incorrect shared memory programs compared to message-passing programs.* To test this hypothesis we can measure the number of students who submit incorrect solutions.

Hyp. 4: *An MPI implementation will require more code than an OpenMP implementation.* To test this hypothesis we can measure the size of code for each implementation.

The classroom studies are the first part of a larger series of studies we are conducting (Figure 2). We first run pilot studies with students. We next conduct classroom studies, and then move onto controlled studies with experienced programmers, and finally conduct experiments in situ with development teams. Each of these steps contributes to our testing of hypothesizes by exploiting the unique aspects of each environment (i.e., replicated experiments in classroom studies and multi-person development with in situ teams). We can also compare our results with recent studies of existing HPC codes [PO05].

2.1 Collection of folklore data

One of the main goals of the development time working group of HPCS project is to leverage HPC community's knowledge of development time issues. In order to do so, we are soliciting expert opinion on issues related to HPC programming by collecting elements of folklore through surveys, generating discussion among experts on these elements of the lore to increase precision of statements and to measure degree of consensus and finally generate testable hypotheses based on the lore that can be evaluated in empirical studies.

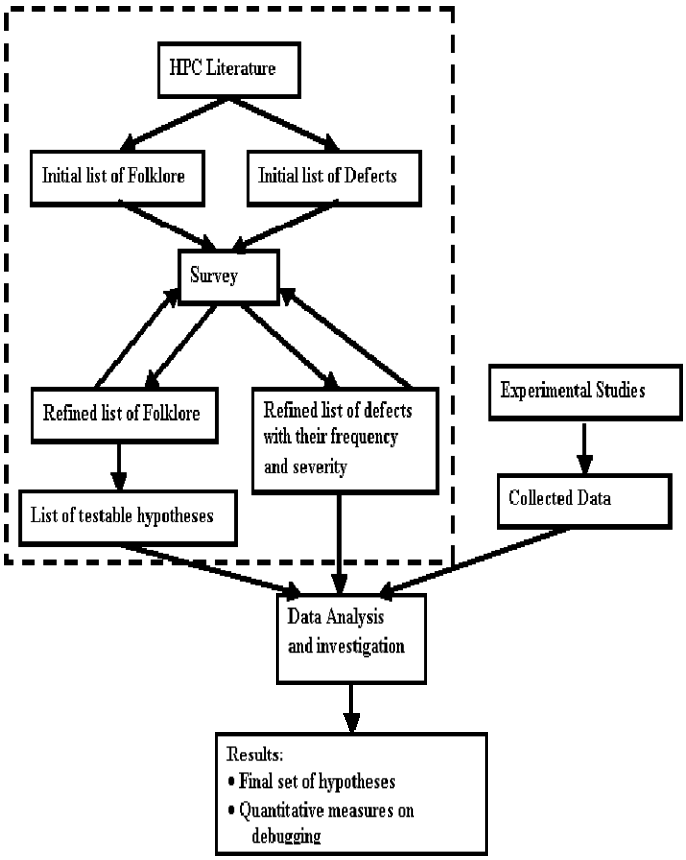

Figure 3: Folklore and defect solicitation process

Before starting the exploratory experiment of collecting peoples' anecdotal beliefs through surveys, we needed an initial set of such anecdotes to both encourage thinking and also use as examples of what we are interested in.

To gather the folklore in HPC, a member of the study group, who is an HPC professor, conducted an informal scan of several sources including lecture notes used in introductory HPC classes at the University of Maryland as well as scanning the Internet for related keywords (including "HPC tribal lore" and "HPC folklore"). The goal of this process was not to be exhaustive, but instead to gather a sense of the type of information that a beginning HPC programmer might find. This initial list of 10 ideas (the left column of the table in Appendix 1) was recorded and used as the basis for our first survey.

We then asked 7 HPC specialists and professors who regularly teach HPC classes to comment on the initial list. They were asked to give an "agree", "disagree" or "don't know" answer to each candidate, give their comments or change suggestions and add any folklore that they are aware of but is not on the list.

The folklore number 11 in Appendix 1 was added by one of the participants at this stage. Generally the comments revolved around clarifying the domain to which the bit of lore applied. For example was

the bit of lore talking about a user programming model such as OpenMP or hardware architecture such as a multi-threaded machine.

In order to clarify the questionable points we scheduled a discussion session among the participants. This discussion resulted in some modifications in the way folklore sentences were phrased. The right column of the table in Appendix 1 is the result of this modification.

At some point during the discussion, the participants agreed that "MPI programs don't run well when you use lots of small messages because you get latency-limited". In order to include this in the folklore list, the lore number 12 was added to the list.

At the next step of the study, a survey form was compiled from the current list of 12 folklore and distributed to the participants at the "High Productivity Computing Systems, Productivity Team Meeting" held in January 2005. In order to avoid any bias, some of the randomly selected lore were rephrased to imply the logically inverse sentence. Two sets of survey forms were compiled and distributed randomly.

2.2 Defect studies

As part of our effort to understand development issues, our classroom experiments have moved beyond effort analysis and have started to look at the impact of defects (e.g. incorrect or excessive synchronization, incorrect data decomposition) on the development process. By understanding how, when, and the kind of defects that appear in HPC codes, tools and techniques can be developed to mitigate these risks to improve the overall workflow. As we have shown [Ho05], automatically determining workflow is not precise, so we are working on a mixture of process activity (e.g., coding, compiling, executing) with source code analysis techniques. The process of defect analysis we are building consists of the following main activities:

- **Analysis**:
  1. Analyze successive versions of the developing code looking for patterns of changes represented by successive code versions (e.g., defect discovery, defect repair, addition of new functionality).
  2. Record the identified changes.
  3. Develop a classification scheme and hypotheses.

  For example, a small increase in source code, following a failed execution, following a large code insertion could represent the pattern of the programming adding new functionality followed by a test and then defect correction. Syntactic tools that find specific defects can be used to aid the human-based heuristic search for defects.
- **Verification**:

We then need to analyze these results at various levels. Verification consists of the following steps, among others:

1. If we can somehow obtain the "true" defect sets, we can directly compare our analysis results with them to evaluate the analysis results quantitatively.
2. Multiple analysts can independently analyze the source code and record identified defects.
3. Examine individual instances of defects to check if each defect is correctly captured and documented.
4. Provide defect instances and classify them into one of the given defect types. This can be used to check the consistency of the classification scheme.

[Na06] explores this defect methodology in greater detail.

3 Experimental Results

An early result needed to validate our process was to verify that students could indeed produce good HPC codes and that we could measure their increased performance. Table 2 is one set of data that shows that students achieved speedups of approximately 3 to 7 on an 8-processor HPC machine. (CxAy means class number x, assignment number y. This coding was used to preserve anonymity of the student population.)

Data set	Programming Model	Speedup on 8 processors
Speedup measured relative to serial version:		
C1A1	MPI	mean 4.74, sd 1.97, n=2
C3A3	MPI	mean 2.8, sd 1.9, n=3
C3A3	OpenMP	mean 6.7, sd 9.1, n=2
Speedup measured relative to parallel version run on 1 processor:		
C0A1	MPI	mean 5.0, sd 2.1, n=13
C1A1	MPI	mean 4.8, sd 2.0, n=3
C3A3	MPI	mean 5.6, sd 2.5, n=5
C3A3	OpenMP	mean 5.7, sd 3.0, n=4

Table 2: Mean, standard deviation, and number of subjects for computing speedup on Game of Life program.

Measuring productivity in the HPC domain is part of understanding HPC workflows; however, what does productivity mean in this domain [IJ04]? The following is one model that we can derive from the fact that the critical component of HPC programs is the speedup achieved by using a multiprocessor HPC machine over a single processor [Ze05]:

$$\text{Speedup} = \frac{\text{Reference Execution Time}}{\text{Parallel Execution Time}}$$

$$\text{Productivity} = \frac{\text{Relative Speedup}}{\text{Relative Effort}} = \frac{\rho}{1/\varepsilon} = \rho \times \varepsilon$$

$$\text{Relative Effort} = \frac{\text{Parallel Effort}}{\text{Reference Effort}}$$

Productivity is defined as the relative speedup of a program using an HPC machine compared to a single processor divided by the relative effort to produce the HPC version of the program divided by the effort to produce a single processor version of the program.

Program →	**1**	**2***	**3**	**4**	**5**
Serial effort (hrs)	3	7	5	15	
Total effort (hrs)	16	29	10	34.5	22
Serial Exec (sec)	123.2	75.2	101.5	80.1	31.1
Parallel Exec (sec)	47.7	15.8	12.8	11.2	8.5
Speedup	1.58	4.76	5.87	6.71	8.90
Relative Effort	2.29	4.14	1.43	4.93	3.14
Productivity	0.69	1.15	4.11	1.36	2.83
*- Reference serial implementation					

Table 3. Productivity experiment: Game of Life

Table 3 shows the results for one group of students programming the Game of life (a simple nearest neighbor cellular automaton problem where the next generation of "life" depends upon surrounding cells in a grid and a popular first parallel program for HPC classes) [Ga70]. The data shows that our definition of productivity had a negative correlation compared to both total effort and HPC execution time, and a positive correlation compared to relative speedup. While the sample size is too small for a test of significance, the relationships all indicate that productivity does behave as we would want a productivity measure to behave for HPC programs, i.e., good productivity means lower total effort, lower HPC execution time and higher speedup.

	Serial	MPI	OpenMP	Co-Array Fortran	StarP	XMT
Nearest-Neighbor Type Problems						
Game of Life	C3A3	C3A3 C0A1 C1A1	C3A3			
Grid of Resistors	C2A2	C2A2	C2A2		C2A2	
Sharks & Fishes		C6A2	C6A2	C6A2		
Laplace's Eq.		C2A3			C2A3	
SWIM			C0A2			
Broadcast Type Problems						
LU Decomposition			C4A1			
Parallel Mat-vec					C3A4	
Quantum Dynamics		C7A1				
Embarrassingly Parallel Type Problems						
Buffon-Laplace Needle		C2A1 C3A1	C2A1 C3A1		C2A1 C3A1	
Other						
Parallel Sorting		C3A2	C3A2		C3A2	
Array Compaction						C5A1
Randomized Selection						C5A2

Table 4. Some of the early classroom experiments on specific architectures.

Table 4 shows the distribution of programming assignments across different programming models for the first 7 classes (using the same CxAy coding used in Table 2). Multiple instances of the same programming assignment lend the results to meta-analysis to be able to consider larger populations of students. Table 5 summarizes the number of times each technology has been applied to each programming problem. More details are given in [Ho05] and [Al07].

Problem	serial	MPI	OpenMP	Matlab*P	XMT-C	Co-Array Fortran	UPC	Hybrid MPI-OpenMP
Game of life	4	5	2	1		2	2	
SWIM			1					
Buffon-Laplace	2	3	2	3				
Laplace's equation	1	1	1	1				
Sharks & fishes	1	2	2			1		
Grid of resistors	1	1	1	1				
Matrix power via prefix		3	1			1	1	
Sparse conjugate-gradient		2				1	1	
Dense matrix-vector multiply	1	1	1					
Sparse matrix-vector multiply	1	1			2			
Sorting	2	3	1		2			
Quantum dynamics		2						
Molecular dynamics								1
Randomized selection					1			
Breadth-first search					1			
LU decomposition			1					
Shortest path			1					
Search for intelligent puzzles		1						

Table 5. Multiple classrooms studies for each technology.

Dataset	Programming Model	Application	Lines of Code
C3A3	Serial	Game of Life	mean 175, sd 88, n=10
	MPI		mean 433, sd 486, n=13
	OpenMP		mean 292, sd 383, n=14
C2A2	Serial	Resistors	42 (given)
	MPI		mean 174, sd 75, n=9
	OpenMP		mean 49, sd 3.2, n=10

Table 6. MPI program size compared to OpenMP program size.

For example, we can use this data to partially answer an earlier stated hypothesis (Hyp. 4: *An MPI implementation will require more code than an OpenMP implementation*). Table 6 shows the relevant data giving credibility to this hypothesis (but this data is not statistically significant yet).

1. An alternative parallel programming model is the PRAM model, which supports fine-grained parallelism and has a substantial history of algorithmic theory [Vi98]. XMT-C is an extension of the C language that supports parallel directives to provide a PRAM-like model to the programmer. A prototype compiler exists that generates code which runs on a simulator for an XMT architecture. We conducted a feasibility study in a class to compare the effort required to solve a particular problem. After comparing XMT-C development to MPI, on average, students required less effort to solve the problem using XMT-C compared to MPI. The reduction in mean effort was approximately 50%, which was statistically significant at the level of $p<.05$ using a t-test [Ho06].
2. While OpenMP generally required less effort to complete (Figure 4), the comparison of defects between MPI and OpenMP, however, did not yield statistically significant results, which contradicted a common belief that shared memory programs are harder to debug. However, our defect data collection was based upon programmer-supplied effort forms, which we know are not very accurate. This led to the defect analysis mentioned previously [Na06], where we intend to do a more thorough analysis of defects made.

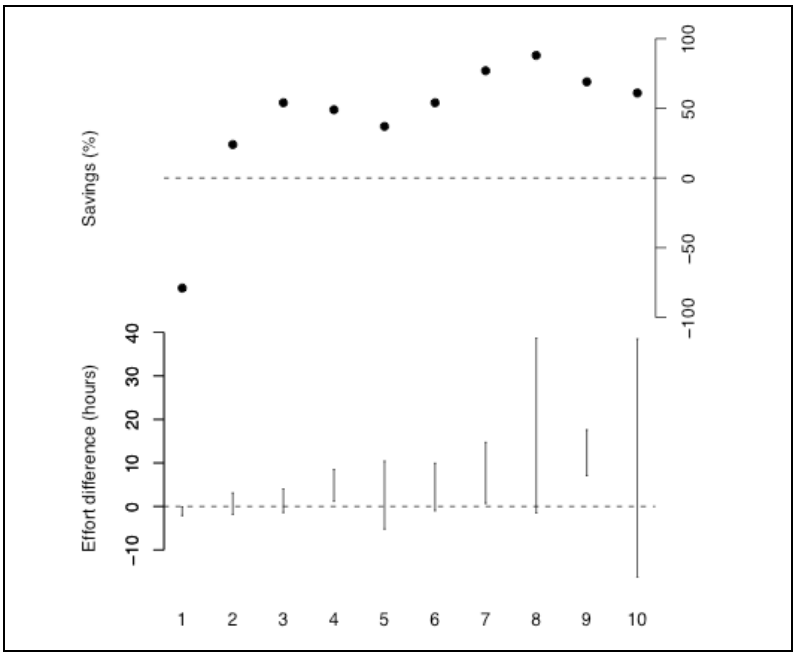

Figure 4. Time saved using OpenMP over MPI for 10 programs. (MPI used less time only in case 1 above).

3. We are collecting low-level behavioral data from developers in order to understand the "workflows" that exist during HPC software development. A useful representation of HPC workflow could both help characterize the bottlenecks that occur during development and support a comparative analysis

of the impact of different tools and technologies upon workflow. One hypothesis we are studying is that the workflow can be divided into one of five states: serial coding, parallel coding, testing, debugging, and optimization.

In a pilot study at the University of Hawaii in Spring of 2006, students worked on the Gauss-Seidel iteration problem using C and PThreads in a development environment that included automated collection of editing, testing, and command line data using Hackystat. We were able to automatically infer the "serial coding" workflow state as the editing of a file not containing any parallel constructs (such as MPI, OpenMP, or PThread calls), and the "parallel coding" workflow state as the editing of a file containing these constructs. We were also able to automatically infer the "testing" state as the occurrence of unit test invocation using the CUTest tool. In our pilot study, we were not able to automatically infer the debugging or optimization workflow states, as students were not provided with tools to support either of these activities that we could instrument.

Our analysis of these results leads us to conclude that workflow inference may be possible in an HPC context. We hypothesize that it may actually be easier to infer these kinds of workflow states in a professional setting, since more sophisticated tool support is often available which can help support inferencing regarding the intent of a development activity. Our analyses also cause us to question whether the five states that we initially selected are appropriate for all HPC development contexts. It may be that there is no "one size fits all" set of workflow states, and that we will need to define a custom set of states for different HPC organizations in order to achieve our goals.

Additional early classroom results are given in [Ho05b].

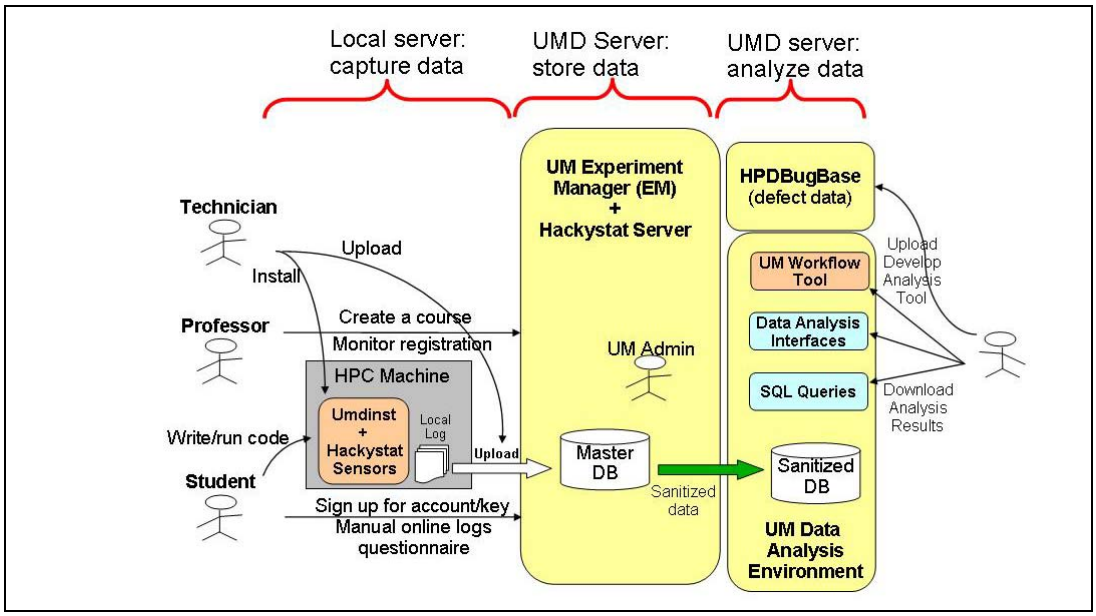

Figure 5. Experiment Manager Structure.

4 Tool Development

In order to conduct this research, a series of tools are being developed:

1. Websites

1. HPCBugBase.org – Defect database
2. http://hpcs.cs.umd.edu – HPCS Development time website

2 Data collection

3. UMDINST – Shell-level time stamps from compilation and execution
4. Experiment Manager - Collect self-reported effort data

5. Shell Logger – Capture all shell commands
6. Hackystat – Low level time stamps for many tools

3 Data conversion

7. Raw data importer – Import UMDINST data to database
8. DB Sanitizer – Remove privacy data from DB

4 Visualization and analysis

9. Automatic Performance Measurement System – Automatically run scripts of programs.
10. UCSB execution harness – Execute programs under controlled conditions
11. CodeVizard – View source code evolution
12. Data Analyzer – Visualization of UMDINST and Experiment Manager data
13. Activity graph – View workflow information

The websites (1.1., 1.2) allow for others to track our research and collect defects via a wiki. The HPCBugBase is described in more details in Section 4.2 below.

The data collection tools (2.3 to 2.6) are focused around the UMD experiment manager, explained in section 4.1 that follows. Hackystat is a product of Philip Johnson of the University of Hawaii, but we have worked with Philip in addressing issues his tool originally didn't process. The data conversion tools (3.7, 3.8) are internal conversion tools within the experiment manager used to handle data from a variety of sources and to handle privacy issues required by federal and state laws. The visualization tools (4.9 through 4.13) are still under development, and the lack of continued funding prevents their completion. A prototype of the data analyzer (tool 4.12) is part of the Experiment Manager explained below.

4.1 Experiment Manager

As stated earlier, we have collected effort data from student developments and begun to collect data from professional HPC programmers in 3 ways: manually from the participants, automatically from timestamps at each system command, and automatically via the Hackystat tool, sampling the active task at regular intervals. All 3 methods provide different values for "effort," and we developed models to integrate and filter each method to provide an accurate picture of effort.

Our collection methods evolved one at a time. To simplify the process of students (and other HPC professionals) providing needed information, we developed an experiment management package (Experiment Manager) to more easily collect and analyze this data during the development process. It includes effort, defect and workflow data, as well as copies of every source program during development. Tracking effort and defects should provide a good data set for building models of productivity and reliability of HEC codes.

We evolved the Experiment Manager framework (Figure 5) to mitigate the complexities described in the previous section. The framework is an integrated set of tools to support software engineering experiments in HPC classroom environments. While aspects of the framework have been studied by others, the integration of all features allows for a uniform environment that has been used in over 25 classroom studies over the past 4 years. The framework supports the following.

1. Minimal disruption of the typical programming process. Study participants solve programming tasks under investigation using their typical work habits, spreading out programming tasks over several days. The only additional activity required is filling out some online forms. Since we do not require them to complete the task in an alien environment or work for a fixed, uninterrupted length of time, we minimize any negative impact on pedagogy or subject overhead.

2. Consistent instruments and artifacts. Use of the framework ensures that the same types of data will be collected and the same types of problems will be solved, which increases confidence in meta-analysis across studies at different universities.

3. Centralized data repository with web interface. The framework provides a simple, consistent interface to the experimental data for experimentalists, subjects, and collaborating professors. This reduces overhead for all stakeholders and ensures that data is consistently collected across studies.

4. Sanitization of sensitive data. The framework provides external researcher with access to the data sets that have been stripped of any information that could identify subjects, to preserve anonymity and

comply with the protocols of human subject research as set out by Institutional Review Boards (IRBs) at American universities.

The Experiment Manager has 3 components:

1. *UMD server*: This web server is the entry portal to the Experiment Manager for students, faculty and analysts and contains the repository of collected data.
2. *Local server*: A local server is established on the user machine (e.g., the one used by students at a university) that is used to capture experimental data before transmission to the University of Maryland.
3. *UMD analysis server*: A server stores sanitized data available to the HPCS community for access to our collected data. This server avoids many of the legal hurdles implicit with using human subject data (e.g., keeping student identities private).

   For the near future, our efforts will focus on the following tasks:

   - Evolve the interface to the Experiment Manager web-based tool to simplify use by the various stakeholders (i.e., roles).
   - Continue to develop our tool base, such as the defect data base and workflow models.
   - Build our analysis data base including details of the various hypotheses we have studied in the past.
   - Evolve our experience bases to generate performance measures for each program submitted in order to have a consistent performance and speedup measure for use in our workflow and time to solution studies.

Further details of the Experiment Manager are covered in [Ho08].

4.2 HPCBugBase

A database and corresponding web-based tool has been created to collect and manage defects found in HPC programs. The tool is at http://www.HPCBugBase.org. It is a public wiki allowing for the creation of a defect-based experience base. Figure 6 is the home page of this bug base. Table 7 presents the initial defect classification taxonomy being developed. More information about the HPC Bug Base is found is [Na06] and [Na07].

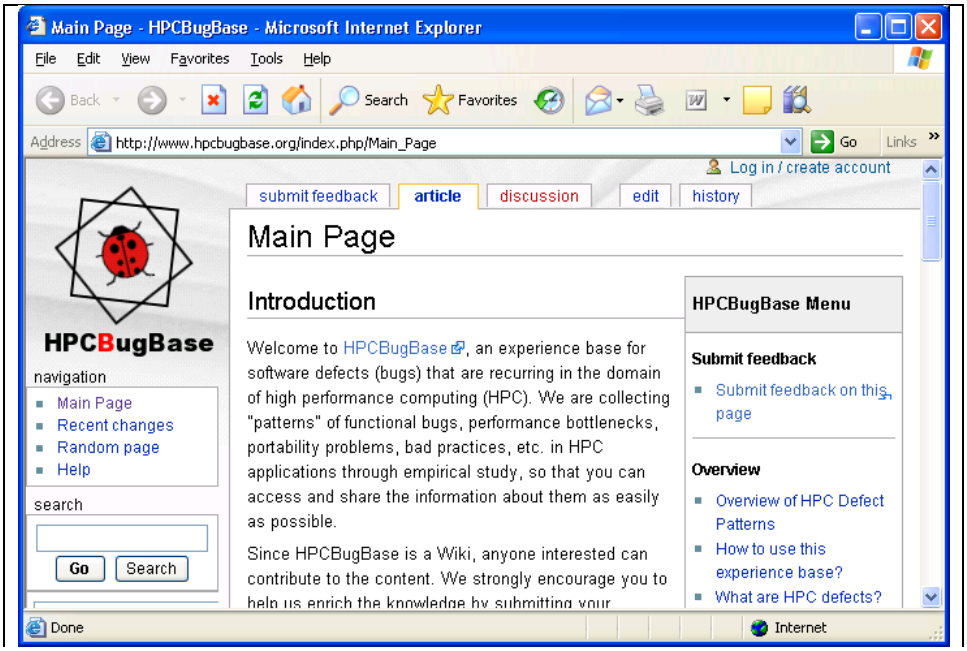

Figure 6. HPCBugBase home page

Top-level defect type	Sub-types	Brief definition
Use of parallel language features	--	Erroneous use of parallel language features
Space decomposition	--	Incorrect mapping between the problem space and the problem memory space
Side-effect of parallelization (hidden serialization)	I/O hotspots	Serial constructs causing correctness and performance defects when accessed in parallel contexts
	Hidden serialization in library functions	
Synchronization	Deadlock	Incorrect/unnecessary synchronization
	Race	
Performance	Message scheduling	Performance defects in parallel contexts
	Load balancing	
	Failure to exploit locality	
	Excessive synchronization	
	Communication/computation ratio	
	Scalability problems	
	Memory hierarchy	
Memory management	Memory allocation	Inadequate memory management
	Memory cleanup	
Algorithm	--	Program logic not matching the intended purpose of the code
Environment	--	Failure to understand/use code written by other people (library code)

Table 7. HPC Defect classification

5 Conclusions

5.1 Student research

Many students have been supported by this research. One MS degree was partially supported: Rola Alameh, 2007; as well as 3 PhD dissertations were partially supported: Lorin Hochstein, 2006, Taiga Nakamura, 2007, and Daniela Cruzes, 2007.

5.2 Publications

The following publications were partially sponsored by this grant:

1. Zelkowitz M., V. Basili, S. Asgari, L. Hochstein, J. Hollingsworth, and T. Nakamura, Productivity measures for high performance computers, Computer Society International Symposium on Software Metrics, Como, Italy, September, 2005.
2. Hochstein L., V. Basili, M. Zelkowitz, J. Hollingsworth and J. Carver, Combining self-reported and automatic data to improve effort measurement, Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2005), Portugal, September.
3. Hochstein L., J. Carver, F. Shull, S. Asgari, V. Basili, J. K. Hollingsworth, M. Zelkowitz, HPC Programmer Productivity: A Case Study of Novice HPC Programmers, Supercomputing 2005, Seattle, WA, November 2005 .

4. Shull F., J. Carver, L. Hochstein, and V. Basili, Empirical study design in the area of high performance computing (HPC), Computer Society International Symposium on Empirical Software Engineering, Noosa Heads, Australia, November, 2005, 305-314.
5. Lorin Hochstein, Victor R. Basili, "An Empirical Study to Compare Two Parallel Programming Models". 18th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '06). July 2006, Cambridge, MA.
6. Taiga Nakamura, Lorin Hochstein, Victor R. Basili, "Identifying Domain-Specific Defect Classes Using Inspections and Change History", Proceeding of 5th ACM-IEEE International Symposium on Empirical Software Engineering (ISESE'06), September 21-22, 2006, Rio de Janeiro, Brazil.
7. Lorin Hochstein, Development of an empirical approach to building domain-specific knowledge applied to high-end computing, PhD dissertation, University of Maryland, Computer Science, August 2005.
8. Jeffrey C. Carver, Lorin Hochstein, Richard P. Kendall, Taiga Nakamura, Marvin V. Zelkowitz, Victor R. Basili and Douglass E. Post, Observations about Software Development for High End Computing, CTWatch, November 2006.
9. Lorin Hochstein, Taiga Nakamura, Victor R. Basili, Sima Asgari, Marvin V. Zelkowitz, Jeffrey K. Hollingsworth, Forrest Shull, Jeffrey Carver, Martin Voelp, Nico Zazworka, Philip Johnson, Experiments to Understand HPC Time to Development, CTWatch, November 2006.
10. Nicole Wolter, Michael O. McCracken, Allan Snavely, Lorin Hochstein, Taiga Nakamura and Victor Basili, What's working in HPC: Investigating HPC User Behavior and Productivity, CTWatch (2)4A:9-17, November, 2006.
11. Andrew Funk, Victor Basili, Lorin Hochstein and Jeremy Kepner, Analysis of parallel software development using relative development time productivity metric, CTWatch (2)4A:46-51, November 2006.
12. Rola Alameh, Nico Zazworka, Jeffrey K. Hollingsworth, Performance Measurement of Novice HPC Programmers' Code, Workshop on Software Engineering of High Productivity Computers 2007, May 2007, Minneapolis, MN.
13. Taiga Nakamura, Recurring software defects in high end computing, PhD dissertation, University of Maryland, Computer Science, May, 2007.
14. Daniela Cruzes, Secondary Analysis on Experimental Software Engineering Studies, PhD dissertation, Salvador University (UNIFACS), Salvador, Brazil, 2007
15. Rola Alameh, Investigating the effects of HPC novice programmer variations on code performance, MS thesis, University of Maryland, Computer Science, December 2007 (to appear).

5.3 Summary

Over the past 3 years we have been developing a methodology for running HPC experiments in a classroom setting and obtaining results we believe are applicable to HPC programming in general. We are starting to look at larger developments and look at large university and government HPC projects in order to increase the confidence on the early results we have obtained with students.

Our development of the Experiment Manager system and HPC bug base allows us to more easily expand our capabilities in this area. This allows many others to run such experiments on their own in a way that allows for the appropriate controls of the experiment so that results across classes and organization at geographically diverse locations can be compared in order to get a thorough understanding of the HPC development model.

6 Acknowledgement

Several additional students worked on various aspects of this project including Patrick R. Borek, Daniela Suares Cruzes, and Thiago Escudeiro Craveiro. We'd also like to acknowledge the following

faculty for allowing us to conduct experiments in their classes: Alan Edelman [MIT], John Gilbert [UCSB], Mary Hall, Aiichiro Nakano, Jackie Chame [USC], Allan Snavely [UCSD], Alan Sussman, Uzi Vishkin, [UMD], Ed Luke [MSU], Henri Casanova [UH], and Glenn Luecke [ISU].

7 References

[As05] S. Asgari, L. Hochstein, V. Basili, M. Zelkowitz, J. Hollingsworth, J. Carver, and F. Shull, Generating Testable Hypotheses from Tacit Knowledge for High Productivity Computing, 2nd International Workshop on Software Engineering for High Performance Computing System Applications, (May, 2005) St. Louis, MO, 17-21.

[Al07] Alameh R., Investigating the effects of HPC novice programmer variations on code performance, MS thesis, University of Maryland, Computer Science, December 2007 (to appear).

[Ba94] V. Basili, S. Green, Software Process Evolution at the SEL, *IEEE Software* 11(4), (July 1994), 58-66.

[Ba97] V. Basili, "Evolving and Packaging Reading Technologies," *Journal of Systems and Software*, vol. 38 (1): 3-12, July 1997.

[Ba99] V. Basili, F. Shull, and F. Lanubile, "Building Knowledge through Families of Experiments," *IEEE Transactions on Software Engineering*, vol. 25(4): 456-473, July 1999.

[Ba02] V. Basili, F. McGarry, R. Pajerski, M. Zelkowitz, Lessons learned from 25 years of process improvement: The rise and fall of the NASA Software Engineering Laboratory, IEEE Computer Society and ACM International Conf. on Soft. Eng., Orlando FL, May 2002, 69-79.

[Ca03] J. Carver, L. Jaccheri, S. Morasca, F. Shull, Issues in using students in empirical studies in software engineering education, International Symposium on Software Metrics, Sydney, Australia, (2003), 239-249.

[Ga70] M. Gardner, Mathematical games, Scientific American, October, 1970.

[Ho05] L. Hochstein, V. Basili, M. Zelkowitz, J. Hollingsworth and J. Carver, Combining self-reported and automatic data to improve effort measurement, Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2005), Lisbon, Portugal, September 2005, 356-365.

[Ho05b] L. Hochstein, J. Carver, F. Shull, S. Asgari, V. Basili, J. K. Hollingsworth, M. Zelkowitz, HPC Programmer Productivity: A Case Study of Novice HPC Programmers, Supercomputing 2005, Seattle, WA, November 2005.

[Ho06] L. Hochstein, V. R. Basili, An Empirical Study to Compare Two Parallel Programming Models, 18th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '06). July 2006, Cambridge, MA.

[Ho08] L. Hochstein, T. Nakamura, F. Shull, N. Zazaworka, V. Basili and M. Zelkowitz, An Environment for Conducting Families of Software Engineering Experiments, Advances in Computers, Elsevier, Boston MA vol. 73 (2008) (to appear)

[IJ04] The International Journal of High Performance Computing Applications, (18)4, Winter 2004.

[Jo04] P. M. Johnson, H. Kou, J. M. Agustin, Q. Zhang, A. Kagawa and T. Yamashita, Practical automated process and product metric collection and analysis in a classroom setting: Lessons learned from Hackystat-UH, Proceedings of the 2004 International Symposium on Empirical Software Engineering, Los Angeles, California, August, 2004.

[Mi00] J. Miller, Applying meta-analytical procedures to software engineering experiments, *Journal of Systems and Software* 54, 1, (September, 2000) 29-39.

[Na06] T. Nakamura, L. Hochstein and V. R. Basili, Identifying Domain-Specific Defect Classes: Using Inspections and Change History, International Symposium on Empirical Software Engineering, (ISESE), Rio de Janeiro, September, 2006.

[Na07] T. Nakamura, Recurring software defects in high end computing, PhD dissertation, University of Maryland, Computer Science, May, 2007.

[Po05] Post, D., Kendall, R.P., and Whitney, E. Case study of the Falcon Project, Second International Workshop on Software Engineering for High Performance Computing Systems Applications,. St. Louis, MO, 2005.

[Sp05] J. Spacco, J. Strecker, D. Hovemeyer, W. Pugh, Software repository mining with Marmoset: an automated programming project snapshot and testing system, Proceedings of the 2005 International Workshop on Mining Software Repositories, St. Louis, Missouri, (2005), 1-5.

[Vi05] U. Vishkin, S. Dascal, E. Berkovich, and J. Nuzman, Explicit Multi-Threading (XMT) Bridging Models for Instruction Parallelism, 10th ACM Symposium on Parallel Algorithms and Architectures (SPAA), 1998.

[Ze05] M. Zelkowitz, V. Basili, S. Asgari, L. Hochstein, J. Hollingsworth and T. Nakamura, Measuring productivity on high performance computers, IEEE Symp. on Software Metrics, Como, Italy, (September 2005).

Appendix: List of HPC folklore

Initial List	Updated List
[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).	[1] Many people use parallel machines primarily for the large amount of memory available (cache or main).
[2] It's hard to create a parallel language that provides good performance across multiple platforms.	[2] It's hard to create a parallel language that provides good performance across multiple platforms
[3] It's easier to get something working in using a shared memory model than message passing.	[3] It's easier to get something working using a shared memory model than message passing.
[4] It's harder to debug shared memory programs due to race conditions involving shared regions.	[4] Debugging race conditions in shared memory programs is harder than debugging race conditions in message passing programs
[5] Explicit distributed memory programming results in programs that run faster since programmers are forced to think about data distribution (and thus locality) issues.	[5] 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
[6] In master/worker parallelism, the master soon becomes the bottleneck and thus systems with a single master will not scale.	[6] In master/worker parallelism, a system with a single master has limited scalability because the master becomes a bottleneck.
[7] Overlapping computation and communication can result in at most a 2x speedup in a program.	[7] In MPI programs, overlapping computation and communication (non-blocking) can result in at most a 2x speedup in a program.
[8] HPF's data distribution process is also useful for SMP systems since it makes programmers think about locality issues.	[8] For large-scale shared memory systems, you can achieve better performance using global arrays with explicit distribution operations than using Open MP.
[9] Parallelization is easy, 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.	[9] Identifying parallelism is hard, but achieving performance is easy.
[10] It's easy to write slow code on fast machines.	[10] It's easy to write slow code on fast machines. Generally, the first parallel implementation of a code is slower than its serial counterpart.
[11] Experts often start with incorrect programs that capture the core computations and data movements. They get these working at high performance first, and then they make the code functionally correct later.	[11] Sometimes, a good approach for developing parallel programs is to program for performance before programming for correctness.
[12] N/A	[12] Given a choice, it's better to write a program with fewer large messages than many small messages