Experiments to Understand HPC Time to Development

Lorin Hochstein
University of Nebraska

Taiga Nakamura
University of Maryland, College Park

Victor R. Basili
University of Maryland, College Park
Fraunhofer Center, Maryland

Sima Asgari
University of Maryland, College Park

Marvin V. Zelkowitz
University of Maryland, College Park
Fraunhofer Center, Maryland

Jeffrey K. Hollingsworth
University of Maryland, College Park

Forrest Shull
Fraunhofer Center, Maryland

Jeffrey Carver
Mississippi State University

Martin Voelp
University of Maryland, College Park

Nico Zazworka
University of Maryland, College Park

Philip Johnson
University of Hawaii

1. Introduction

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 universities across the U.S. in graduate level HPC courses (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.

Limitations of student studies include the relatively short programming assignments 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.

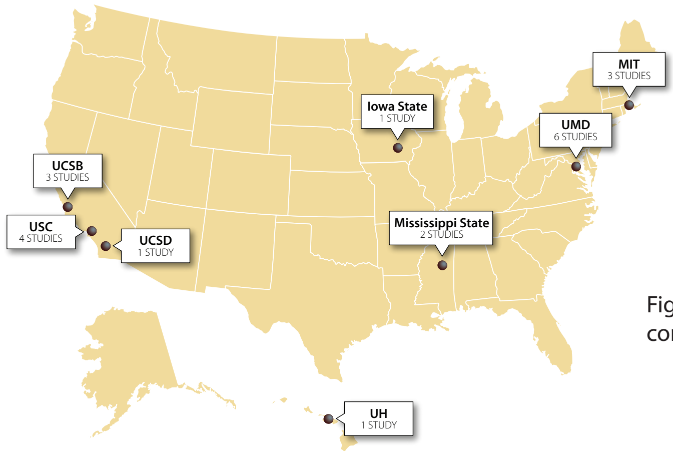

Figure 1. Classroom studies conducted.

In this article, 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), and current plans for improving the process in the future (Section 4).

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 4). 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[1]). 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

[1] Johnson, P. M., Kou, H., Agustin, J. M., Zhang, Q., Kagawa, A., Yamashita, T. "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.

compiler invocations (including capture of source code), all programs invoked by the subject as a shell command, and interactions with supported editors. Elsewhere,[2] we describe the details of how we gather this information and convert it into a record of programmer effort.

[2] Hochstein, L., Basili, V., Zelkowitz, M., Hollingsworth, J., Carver, J. "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.

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:[3]

[3] Asgari, S., Hochstein, L., Basili, V., Zelkowitz, M., Hollingsworth, J., Carver, J., Shull, F. "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.

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, 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 hypotheses 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.[4]

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

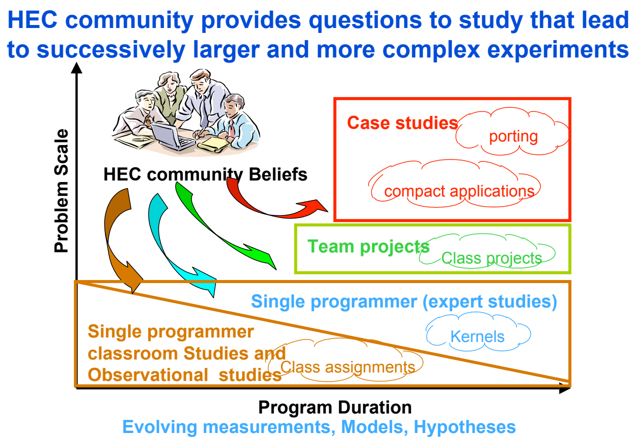
HEC community provides questions to study that lead to successively larger and more complex experiments
Problem Scale
HEC community Beliefs
Case studies
porting
compact applications
Team projects
Class projects
Single programmer (expert studies)
Kernels
Single programmer classroom Studies and Observational studies
Class assignments
Program Duration
Evolving measurements, Models, Hypotheses

Figure 2. Research Plan.

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,[2] 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 and 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.

Nakamura, et al. explore this defect methodology in greater detail.[5]

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

3. Results

Early results needed to validate our process were 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 three to seven 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.

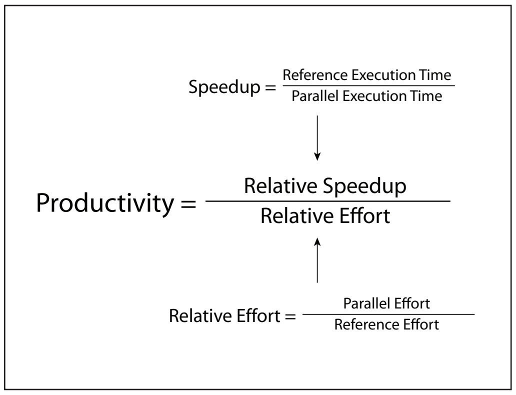
Speedup = Reference Execution Time / Parallel Execution Time
Productivity = Relative Speedup / Relative Effort
Relative Effort = Parallel Effort / Reference Effort

Figure 3. HPC productivity.

Additional classroom results include the following:

1. Measuring productivity in the HPC domain is part of understanding HPC workflows. However, what does productivity mean in this domain?[6] Figure 3 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.[7] 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 compared to a single processor version.

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

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

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).[8] 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.

[8] Gardner, M. "Mathematical games," *Scientific American*, October, 1970.

	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 seven 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.[2]

[2] Hochstein, L., Basili, V., Zelkowitz, M., Hollingsworth, J., Carver, J. "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.

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 5. 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 5 shows the relevant data giving credibility to this hypothesis (but this early data is not statistically significant yet).

2. An alternative parallel programming model is the PRAM model, which supports fine-grained parallelism and has a substantial history of algorithmic theory.[9] XMT-C is an extension of the C language that supports parallel directives to provide a PRAM-like

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

model to the programmer. A prototype compiler exists that generates code that 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.[10]

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

3. 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,[5] where we intend to do a more thorough analysis of defects made.

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

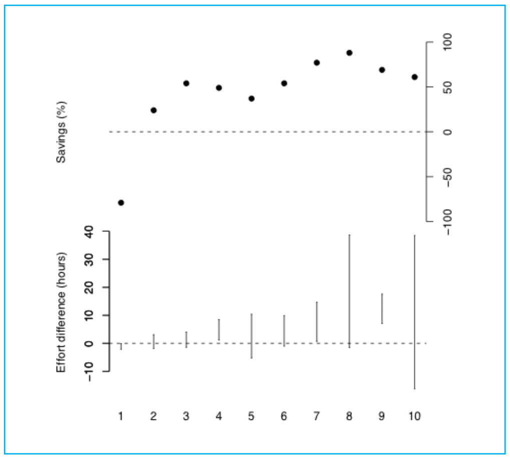

Figure 4. Time saved using OpenMP over MPI for 10 programs. (MPI used less time only in case 1 above).

4. 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 that 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 Hochstein, et al.[11]

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

4. Current Developments

As stated earlier, we have collected effort data from student developments and begun to collect data from professional HPC programmers in three 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 three 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.

The Experiment Manager (pictured in Figure 5) has three components:

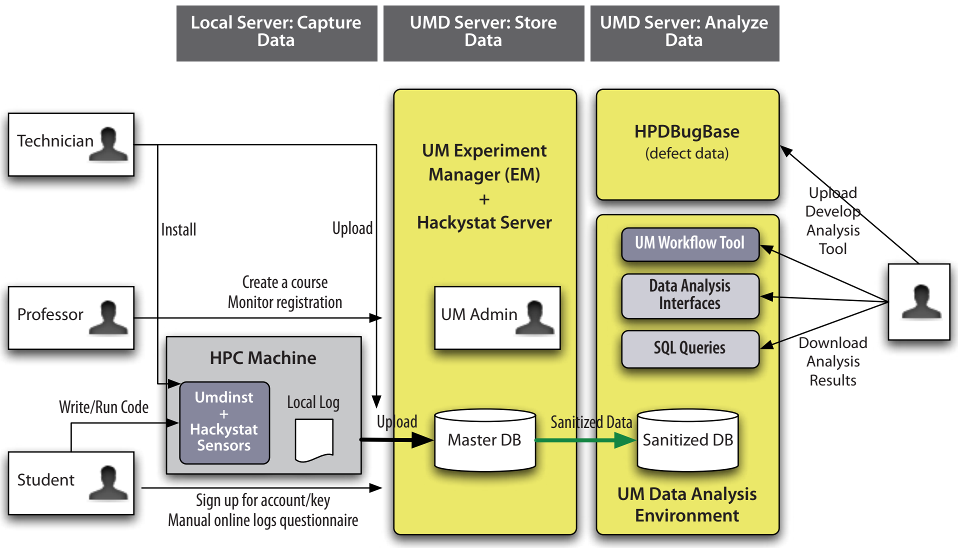
Local Server: Capture Data
UMD Server: Store Data
UMD Server: Analyze Data
Technician
Install
Upload
UM Experiment Manager (EM) + Hackystat Server
HPDBugBase (defect data)
Upload Develop Analysis Tool
UM Workflow Tool
Data Analysis Interfaces
SQL Queries
Professor
Create a course Monitor registration
UM Admin
HPC Machine
Umdinst + Hackystat Sensors
Local Log
Write/Run Code
Upload
Master DB
Sanitized Data
Sanitized DB
Download Analysis Results
Student
Sign up for account/key Manual online logs questionnaire
UM Data Analysis Environment

Figure 5. Experiment Manager Structure.

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.

5. Conclusions

Over the past three 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 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 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.

Acknowledgments
This research was supported in part by Department of Energy contract DE-FG02-04ER25633 and Air Force grant FA8750-05-1-0100 to the University of Maryland.

CTWatch QUARTERLY

ISSN 1555-9874

VOLUME 2 NUMBER 4A **NOVEMBER 2006**

HIGH PRODUCTIVITY COMPUTING SYSTEMS AND THE PATH TOWARDS USABLE PETASCALE COMPUTING

GUEST EDITOR **JEREMY KEPNER**

SC2006 PRINT EDITION

Available on-line at http://www.ctwatch.org/quarterly/

CyberInfrastructure Partnership

Cyberinfrastructure Technology Watch
http://www.ctwatch.org/