Parallel Programmer Productivity: A Case Study of Novice Parallel Programmers

Lorin Hochstein[1], Jeff Carver[3], Forrest Shull[2], Sima Asgari[1], Victor Basili[1,2], Jeffrey K. Hollingsworth[1], Marvin V. Zelkowitz[1,2]

[1]University of Maryland, College Park
{sima,hollings,lorin}@cs.umd.edu

[2]Fraunhofer Center Maryland
{basili,fshull,mvz}@fc-md.umd.edu

[3]Mississippi State University
carver@cse.msstate.edu

Abstract

In developing High-Performance Computing (HPC) software, time to solution is an important metric. This metric is comprised of two main components: The human effort required developing the software, plus the amount of machine time required to execute it. To date, little empirical work has been done to study the first component: the human effort required and the effects of approaches and practices that may be used to reduce it. In this paper, we describe a series of studies that address this problem. We instrumented the development process used in multiple HPC classroom environments. We analyzed data within and across such studies, varying factors such as the parallel programming model used and the application being developed, to understand their impact on the development process.

1 Introduction

Historically, the major metric of success of computation in the HPC community has been speed of program execution. Recently the community has recognized the importance of the time required to develop programs as well as run them. However, since HPC applications must, by definition, be high performance, it is critical to study programmer productivity and application performance together. The goal of this work is to better understand and quantify software development for high performance computers, to augment the existing work on improving performance time, and to take a more complete view of the total time to solution. Currently there is little empirical evidence to support or refute the utility of specific programming models, languages, and practices within the HPC community.

While there is a rich body of literature in the Software Engineering community about programmer productivity, much of it was developed with assumptions that do not necessarily hold in the HPC community. For example, while the SE community expects software specifications to evolve over the lifetime of the system, this evolution is expected to be in response to external factors such as customer-specified requests for new or changed functionality. However, in scientific computation, it is often insights culled from results of one program version that drive the needs for the next. This is because the software itself is helping to push the frontiers of understanding rather than automate well-understood tasks. Due to these unique requirements, traditional software engineering approaches for improving productivity may be adapted for the HPC community, but are not appropriate without changes.

With these challenges in mind, we have been developing and debugging a set of tools and protocols to study programmer productivity in the HPC community. In this paper, we present both the methodology we have developed to investigate programmer productivity issues in the HPC domain, and some initial results of studying productivity of novice HPC programmers.

The interest in the effectiveness of novice developers is justified by the nature of the traditional HPC context. Many of the applications in which high performance computers are used are quite complex and understood only by domain experts. Those domain experts will often be novice HPC programmers, at least when they are beginning their careers. Useful, evidence-based heuristics about how novice HPC developers can best be brought up to speed hold out the promise of being able to address a significant obstacle to the goal of making correct HPC solutions feasible for more problems.

As the first phase of this work, we are beginning with an observational approach, observing HPC code development practices and describing the results in terms of overall effort, performance, and cost. There is little previous empirical work on this topic, so this work will be used to generate a series of well-grounded hypotheses that can then be tested explicitly in later studies.

2 Related Work

Two main components make up the time to solution metric. The first component is the human effort/calendar time required to develop and tune the software. The second component is the amount of machine time required to execute the software to produce the desired result.

Metrics and even predictive models have already been developed for measuring the code performance part of that equation, under various constraints (e.g. [9, 11]). However,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SC|05 November 12-18, 2005, Seattle, Washington, USA
(c) 2005 ACM 1-59593-061-2/05/0011…$5.00

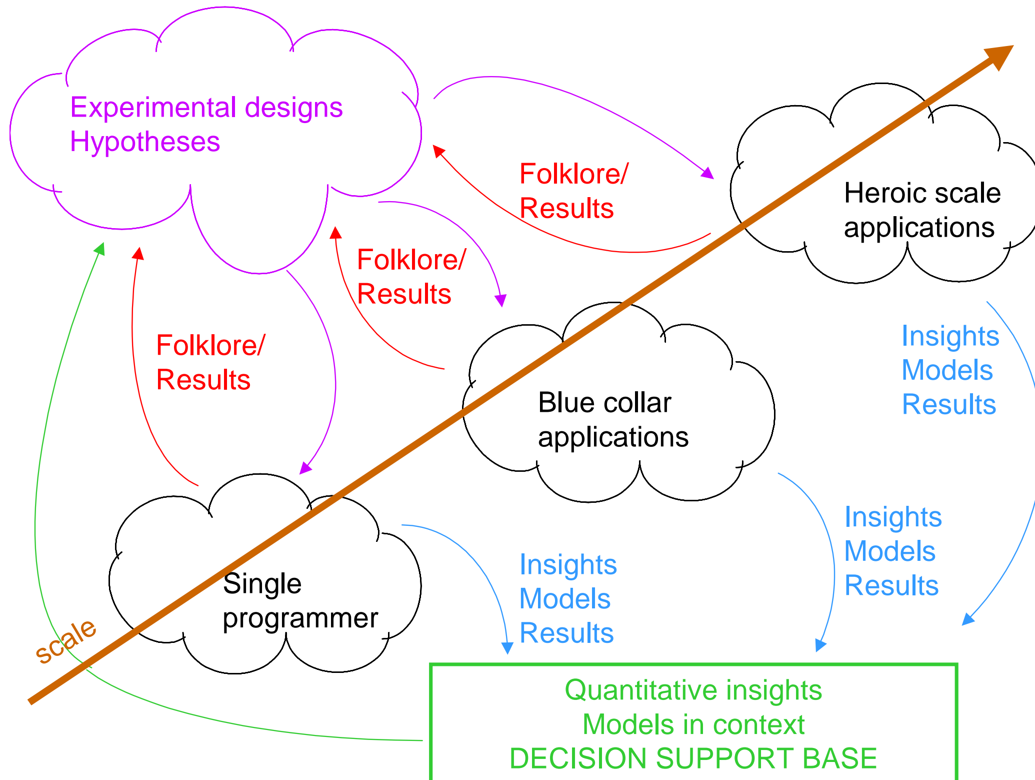

Figure 1: Process of Refining and Evaluating HPC Programmer Productivity Hypotheses.

little empirical work has been done to date to study the human effort required to implement those solutions. Only a handful of empirical studies have been run to examine factors influencing variables such as development time [2] or the difficulties encountered during HPC development [12]. Some authors have commented that "little work has been done to evaluate HPC systems' usability, or to develop criteria for such evaluations" [12]. As a result, many of the practical decisions about development language and approach are currently made based on anecdote, “rules of thumb,” or personal preference.

Several prior studies[4, 6] have used lines of code as the principal metric to determine effort required to compare parallel programming models with the assumption that fewer lines of code implies less effort. While this metric might have merit, we feel it is important to quantify the effort required to produce different lines of code. Even if it turns out that the technologies that require more code always require more effort, the ratio of lines of code doesn't necessarily equal the ratio of efforts, so you can't use LOCs to do your tradeoff analysis.

Previous work done to study more general software development processes will provide a starting point for this work, but in the end contributes little in the way of directly transferable results due to the significant differences in context. For example, one of the major differences between HPC software development and traditional software development is the amount of effort devoted to tuning HPC code for performance. It is widely believed that more time spent tuning the code will result in shorter execution times. Therefore, understanding the tradeoff between time spent in development and execution time is crucial. For large-scale systems, the extra development time can lead to orders of magnitude reduction in execution time.

3 Methodology

Quantifying programmer productivity is a much more difficult task than measuring program performance. Many external factors such as the inherent difference in the productivity of programmers, to seemingly mundane questions of measuring exactly when a programmer is working on a problem, make it difficult to accurately measure programming effort, and thus productivity. If the goal is to compare different programming models, additional factors such as the programmer’s familiarity with the programming model must be controlled for. Given these sets of challenges, we set out to develop data collection mechanisms and experimental protocols to evaluate programmer productivity in the HPC domain.

It is possible to get interesting and meaningful results about effort by doing human-subject experiments. While there is great variation across subjects, carefully conducted studies can both control for this phenomenon and even quantify it. In many of our studies we conduct within-subject comparisons, which mitigate the variation problem somewhat. Also, in doing these studies when multiple subjects conduct the same exercise, you can actually observe and quantify what the variation across programmers looks like. One of our goals in this project is to quantify the degree of difference in effort required for various programming models and paradigms. When conducting these studies, we specifically seek a variety

of programming ability because we want to quantify not only what is possible for the best programmers, but also quantify the degree to which a specific technique increases the depth of programmers that are able to make effective use of HPC resources.

Our initial studies have been focused on classroom studies of students taking introductory graduate courses in high performance computing. Although it, introduces its own set of challenges, the classroom setting provides several useful features as an environment to conduct this research[5]. First, multiple students are routinely given the same assignment to perform, and thus we are able to conduct experiments in a way to control for the skills of specific programmers. Second, 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. Finally, 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 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.

In each class, we obtained consent from students to be part of our study. The nature of the assignments was left to the individual instructors for each class. However, based on previous communication and discussions as part of this project, many of the instructors happen to use the same assignments. To ensure that the data from the study would not impact students' grades, our protocol quarantined the data collected in a class from professors and teaching assistants for that class until final grades had been assigned.

To conduct these studies, we needed a way to measure how much time students spent working on programming assignments, and some idea of what they were doing during this time (e.g. serial coding, parallelization, debugging, tuning). Historically, there have been two ways used to gather this type of information: explicit recording by subject in diaries (either paper or web-based) and implicit recording by instrumenting the development environment. Both of these approaches have strengths and limitations. After conducting a series of tests using variations on these two techniques, we have settled on a hybrid approach that combines diaries with an instrumented programming environment that captures a time-stamped record of every program the student compiles along with the program compiled. In another paper[8], we describe the details of how we gather this information and convert it into a record of programmer effort.

The classroom studies are the first part of a larger series of studies we are conducting as part of this project. Figure 1 shows the process we are using. It is an iterative process that starts by extracting hypotheses from folklore among the HPC community. We then develop ways to test these hypotheses and run pilot studies. We next conduct classroom studies, and then move onto controlled studies with experienced programmers, and finally conduct experiments with in situ studies 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).

In addition to hypotheses about the effort required, we are also studying the workflow of programmers as the program. For example, do programmers start with a serial version of the program and then proceed to a parallel program? Do programmers code, debug for correctness, and then proceed to tune for performance. Like many aspects of HPC development, there is a wealth of beliefs about what programmers do (or should do), but relatively little in the way of studies to validate these beliefs.

4 Description of the Studies

To achieve some coverage of the various problem types and HPC solution approaches, a collaborative, consisting of several universities including the University of Maryland, the University of Southern California, the University of California Santa Barbara, and the Massachusetts Institute of Technology, is conducting a series of classroom studies. Table 1 provides an overview of the problem space and the studies run to date. The values along the dimensions of the matrix are explained further in the following sections. Each individual study is labeled as "CxAy," with codes corresponding to the course (C), a specific class/semester pair (x,y) and the assignment (A).

4.1 Independent Variables

By replicating multiple variations on assignments at different universities, we can vary the *type of application* being developed (vertical axis in Table 1). The 10 assignments given as classroom assignments so far can be grouped into 4 distinct problem types based on their communication requirements: nearest neighbor, broadcast, embarrassingly parallel, and miscellaneous. As we accumulate more data, we will be able to investigate whether there are characteristic patterns observable within and among the various problem types.

The data sets described in this paper were generated for two specific applications: The "game of life" posits a two-dimensional grid where every cell can be either on or off; over a series of turns the grid evolves with the behavior of each cell determined according to a set of rules about the state of the cells surrounding it. The "grid of resistors" requires the computation of the voltage and effective resistance of a 2n+1 by 2n+1 grid of 1 ohm resistors with a battery connected to the two center points.

A second key factor that will vary across assignments is the *Parallel Programming Model* (e.g., message passing or shared memory). The two instances of parallel programming models about which we have collected the most data are MPI (message passing) and OpenMP (shared memory). Some of the researchers whose classes we observed are also using new HPC approaches which they have developed. These include StarP, a parallel extension to the Matlab environment [10], and Explicit Multi-Threading (XMT), a conceptual framework with language extensions to C that implement parallel random-access algorithms [13].

	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			P2A3	
SWIM			C0A2			
Broadcast Type Problems						
LU Decomposition			C4A1			
Parallel Mat-vec					C3A4	
Quantum Dynamics		C7A1				
Embarrassingly Parallel Type Problems						
Buffon-Laplace Nee-dle		C2A1 C3A1	C2A1 C3A1		C2A1 C3A1	
(Miscellaneous Problem Types)						
Parallel Sorting		C3A2	C3A2		C3A2	
Array Compaction						C5A1
Randomized Selection						C5A2

Table 1: Matrix describing the problem space of HPC studies being run. Columns show the parallel programming model user. Rows shows the assignment, grouped by communication pattern required. Each study is indicated with a label CxAy, identifying the participating class (C) and the assignment (A). Studies analyzed in this paper are grey-shaded.

A final important independent variable is *programmer experience.* In the studies reported here, the majority of subjects were novice HPC developers (not surprisingly, as the studies were run in a classroom environment).

4.2 Dependent Variables

Our studies measured the following as outcomes of the HPC development practices applied:

A primary measure of the quality of the HPC solution was the *speedup* achieved, that is, the relative execution time of a program running on a multi-processor system compared to a uniprocessor system. In this paper, all values reported for speedup were measured when the application was run on 8 parallel processors, as this was the largest number of processors that was feasible for use in our classroom environments.

A primary measure of cost was the amount of *effort* required to develop the final solution, for a given problem and given approach. The effort undertaken to develop a serial solution included the following activities: thinking/planning the solution, coding, and testing. The effort undertaken to develop a parallel solution included all of the above as well as tuning the parallel code (i.e. improving performance through optimizing the parallel instructions). HPC development for the studies presented in this paper was always done having a serial version available.

Another outcome variable studied was the *code expansion factor* of the solutions. In order to take full advantage of the parallel processors, HPC codes can be expected to include many more lines of code (LOC) than serial solutions. For example, message-passing approaches such as MPI require a significant amount of code to deal with communication across different nodes. The expansion factor is the ratio of LOC in a parallel solution to LOC in a serial solution of the same problem.

Finally, we look at the *cost per LOC* of solutions in each of the various approaches. This value is another measure (in person-hours) of the relative cost of producing code in each of the HPC approaches.

4.3 Studies Described in this Paper

To validate our methodology, we selected a subset of the cells in the table for which we have already been able to gather sufficient data to draw conclusions (i.e., the gray-shaded cells in Table 1), where the majority of our data lies at this time

Studies analyzed in this paper include:

C0A1. This data was collected in Fall 2003, from a graduate-level course with 16 students. Subjects were asked to implement the "game of life" program in C on a cluster of PCs, first using a serial solution and then parallelizing the solution with MPI.

C1A1. This data was from a replication of the C0A1 assignment in a different graduate-level course at the same university in Spring 2004. 10 subjects participated.

C2A2. This data was collected in Spring 2004, from a graduate-level course with 27 students. Subjects were asked to parallelize the "grid of resistors" problem in a combination of Matlab and C on a cluster of PCs. Given a serial version, sub-

jects were asked to produce an HPC solution first in MPI and then in OpenMP.

C3A3. This data was collected in Spring 2004, from a graduate-level course with 16 students. Subjects were asked to implement the "game of life" program in C on a cluster of PCs, first as a serial and then as an MPI and OpenMP version.

5 Hypotheses Investigated

The data collected from the classroom studies allow us to address a number of issues about how novices learn to develop HPC codes, by looking within and across cells on our matrix. In Sections 5.1-5.4 we compare the two parallel programming models, MPI and OpenMP, to serial development to derive a better understanding of the relationship between serial development and parallel development. Then in Section 5.5 we compare MPI and OpenMP to each other to get a better understanding of their relationship with regard to programmer productivity.

In the analysis presented in Sections 5.2-5.5, we used the paired t-test [7] to compare MPI to OpenMP. For example, we used a paired t-test to investigate whether there was any difference in the LOC required to implement the same solution in different HPC approaches for the same subject. This statistical test calculates the signed difference between two values for the same subject (e.g. the LOC required for an OpenMP implementation and an MPI implementation of the same problem). The output of the test is based on the mean of the differences for all subjects: If this mean value is large it will tend to indicate a real and significant difference across the subjects for the two different approaches. Conversely, if the mean is close to zero then it would tend to indicate that both approaches performed about the same (i.e. that there was no consistent effect due to the different approaches). By making a within-subject comparison we avoid many threats to validity, by holding factors such as experience level, background, general programming ability, etc., constant.

5.1 Achieving Speedup

A key question was whether novice developers could achieve speedup at all. It is highly relevant for characterizing HPC development, because it addresses the question of whether the benefits of HPC solutions can be widely achieved or will only be available to highly skilled developers. A survey of HPC experts (conducted at the DARPA-sponsored HPCS project meeting in January 2004) indicated that 87% of the experts surveyed felt that speedup could be achieved by novices, although no rigorous data was cited to bolster this assertion. Based on this information, we posed the following hypothesis:

H1 Novices are able to achieve speedup on a parallel machine

To evaluate this hypothesis, we evaluated speedup in two ways: 1) Comparing the parallel version of the code to a serial version of the same application, or 2) comparing the parallel version of the code run on multiple processors to the same version of the code run on one processor.

For the *game of life* application, we have data from 3 classroom studies for both MPI and OpenMP approaches. These data are summarized in Table 2, which shows the parallel programming model, the application being developed, and the programming language used in each study for which data was collected. The OpenMP programs were ran on shared memory machines. The MPI programs were run on clusters.

Data set	**Programming Model**	**Speedup on 8 processors**
Speedup w.r.t. 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 w.r.t. 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. All data sets are for C implementations of the Game of Life.

Although there are too few data points to say with much rigor, the data we do have supports our hypothesis H1. Novices seem able to achieve about 4.5x speedup (on 8 processors) for the game of life application over the serial version. Speedup on 8 processors is consistently about 5 compared to the parallel version run on one processor, for this application.

In addition to the specific hypothesis evaluated, we also observed that OpenMP seemed to result in speedup values near the top of that range, but too few data points are available to make a meaningful comparison and too few processors were used to compare the parallelism between programming models.

5.2 Code expansion factor

Although the number of lines of code in a given solution is not important on its own terms, it is a useful descriptive measure of the solution produced. Different programming paradigms are likely to require different types of optimization their solution; for example, OpenMP typically seems to require adding only a few key lines of code to a serial program, while MPI typically requires much more significant changes to produce a working solution. Therefore we can pose the following hypotheses:

H2 An MPI implementation will require more code than its corresponding serial implementation

H3 An OpenMP implementation will require more code than its corresponding serial implementation.

We can use the data from both C2A2 and C3A3 (the first and second rows of the matrix) to investigate how the size of the solution (measured in LOC) changes from serial to various HPC approaches. A summary of the datasets is presented in Table 3. (In the C2A2 assignment, students were given an existing serial solution to use as their starting point. It is in-

cluded here to allow comparison with the OpenMP and MPI solutions to the same application.)

Data set	Program Model	Program	LOC
C3A3	Serial	Life	173/90 (10)
C3A3	MPI	Life	433/485 (13)
C3A3	OpenMP	Life	195/154 (13)
C2A2	Serial	Resistors	42 (given)
C2A2	MPI	Resistors	174/75 (9)
C2A2	OpenMP	Resistors	49/3.2 (10)

Table 3: Mean, standard deviation, and number of subjects for computing code size (Lines of Code). All program were written in the C programming language.

For the *game of life* application, the number of LOC for the MPI solution was significantly greater than in the serial version (p = 0.02), but the number of LOC in OpenMP was not significantly greater than the serial (p = 0.06).

For the *grid of resistors* application, the number of SLOC for the MPI solution was significantly greater than in the serial version (p = 0.0004), as was the number of SLOC in the OpenMP version (p = 8.5e-5).

Thus, *H2* is supported for both problem domains, that is, MPI required significantly more code to express the solution than in serial. *H3* was supported in only one of the two domains. OpenMP on average required many fewer SLOC than MPI.

5.3 Effort required

Any study of HPC development productivity will focus on developer effort as a key measure. For any given development project, developer effort will be a significant contributor to the final cost as well as being a useful indicator of the amount of time required before a solution can be available. Furthermore, the code that is written to parallelize an application is generally thought of as being more difficult to write than the underlying serial code. These two points led us to propose the following hypotheses:

H4 A parallel code (MPI or OpenMP) will require more development effort than its underlying serial code

As in Section 5.2, we can use the data from C3A3 (the second row of the matrix) to investigate any differences in the effort subjects required to implement the same solution using various HPC approaches. The data is summarized in Table 4.

Programming Model	Effort (person-hrs)
Serial	mean 4.4, sd 4.3, n=15
MPI	mean 10.7, sd 8.9, n=16
OpenMP	mean 5.0, sd 3.5, n=16

Table 4: Mean, standard deviation, and number of subjects for computing total effort. All data sets are for C implementations of the Game of Life for data set C3A3.

As explained in Section 5.2, we used one-sided paired t-tests to test for differences between the solutions implemented with various HPC approaches by the same subject.

For the *game of life* application in C3A3, the effort required to implement the solution in MPI was significantly greater than required for the serial (p = 0.002), but is not the case for OpenMP (p = 0.3). For MPI, *H4 is supported with a statistically significant result,* although the variation from one subject to another was very high.

5.4 Cost per LOC

The cost per LOC measure is useful for estimating the relative difficulty of various HPC approaches – that is, how much effort is expended per line of the final solution.

Most experts seem to believe that almost any approach to HPC development will result in a higher cost per line of code than serial development of the same problem. (In the January 2004 survey at the DARPA expert meeting, 73% of respondents felt this was a true statement, while 19% disagreed.) Based on this survey result, we posed the following hypothesis:

H5 The cost per line of code for parallel code (MPI or OpenMP) will be greater than the cost per line of code for serial code

We use the effort data from C3A3, described in Section 4.3, to investigate the cost of HPC code in terms of the amount of effort required per line of code. Specifically, we compared the cost to develop a serial version from scratch to the cost to develop an HPC version starting from an existing serial implementation. As before, we rely on the one-sided paired t-test to test the differences in mean values for statistical significance.

Analysis showed that HPC development was in fact statistically significantly more expensive than serial development, both for MPI (p = 0.006) and for OpenMP (p = 0.005) supporting hypothesis *H5*. Note that one sample was excluded from this analysis because for that single case the OpenMP implementation was smaller than the corresponding serial version.

Programming Model	Effort (person minutes/LOC)
Serial	mean 1.8, sd 1.3, number 10
MPI	mean 5.5, sd 4.2, number 9
OpenMP	mean 24.8, sd 21.0, number 9

Table 5: Mean, standard deviation, and number of subjects for computing effort per line of code. All data sets are for C implementations of the Game of Life for data set C3A3.

5.5 Effects of different HPC approaches

A central question of importance to HPC research is, what are the strengths and weaknesses of the given HPC approaches? That is, if an HPC solution is going to be applied, what are the tradeoffs between cost and benefits for different problem types?

Based on the discussions above and the generally held beliefs about the relationship between MPI and OpenMP, we posed the following hypotheses:

H6 *More effort will be required to write an MPI code than will be required to write an OpenMP code for the same application*

H7 *MPI code requires more effort per line of code than OpenMP code*

We use the effort data from C3A3, described in Section 5.3, to investigate the cost of HPC code in terms of the amount of effort required per line of code. (As in 5.3, the HPC effort represents the effort needed to produce a parallel version starting from an existing serial implementation, not to develop an HPC solution from scratch.) Also as before, we rely on the paired t-test to test the differences in mean values for statistical significance.

In evaluating hypothesis *H6*, the data show that *total* MPI effort was significantly greater than *total* OpenMP effort ($p = 0.005$). However, for hypothesis *H7,*the cost per line of code was not significantly greater for MPI than for OpenMP ($p = 0.96$).

If we view the results discussed in the previous sections together, we can draw some deeper conclusions. Recall that in the speedup data, Section 5.1, OpenMP seemed to produce slightly better speedup on average, although there were not enough data points to make a statistically rigorous comparison. This result also shows that the number of LOC in a given code is not a useful proxy for developer effort by itself. That is, the amount of effort that is spent per LOC will vary from one solution to another depending on the HPC approach used.

5.6 Threats to validity

The fact that these studies were run, not only in a classroom environment, but across several classroom environments, means that there are threats to the validity of our conclusions that should be kept in mind when interpreting the results. Here we discuss the internal and external threats to validity for the studies described in this paper [3].

We have already characterized the major context variables that may vary from one classroom environment to another, such as the programming language being used or the specific application assigned to the students. We have argued that the experience level of subjects was rather similar and thus comparable across studies: Before conducting the study, we asked the students how much experience they had in parallel programming. Table 6 shows the distribution of the responses for each class. The distributions suggest that the classes are comparable: nearly all of the students had either no experience or only classroom experience in parallel programming. Therefore, comparing data across classes is less dangerous than if the subject populations had been more heterogeneous.

Class	Prior Experience		
	None	Classroom	Professional
C0	56%	33%	11%
C1	63%	38%	0%
C2	100%	0%	0%
C3	50%	50%	0%

Table 6: Reported parallel programming experience.

In terms of internal validity, we faced two main threats to validity: *instrumentation* and *learning*. We used performance data that was reported by the subjects. This falls into the category of an *instrumentation* threat, since the students may not have reported their data truthfully or consistently across the entire population. They may have also made errors while recording execution time data (e.g. recording execution time when the machine is heavily loaded).

An additional *instrumentation* threat is that all effort data was collected through a combination of automatic time stamp logs, generated by instrumenting the compiler and batch scheduler on the development machines, and of self-reported logs kept by the students (which were emailed directly to our research group rather than to the class instructor, to avoid fear of data collection influencing grades). Our process for reconciling both types of data has been described in some detail elsewhere [1]. However, we should note that it is possible that the serial development effort has been underestimated: While students were constrained to use the HPC machines assigned for OpenMP or MPI development (getting access to parallel machines outside of the classroom environment is highly difficult), students could develop the serial component of an assignment on almost any machine. If students both worked on an un-instrumented machine and under-reported their effort spent on the assignment on the manual forms, we may have under-counted their effort.

Finally, there is the possibility that students experienced a *learning* effect, i.e. that solving the problem using one approach affected the subjects' ability to solve the problem using another approach. The results may be confounded by the order in which the subjects solved the problem using the different programming models. This threat can be addressed by enforcing an ordering on the subjects, where half of them solve the problem in one order and another half solve it in a different

order. Unfortunately, we were not able enforce ordering in these studies.

There are several external threats to validity which are common to software engineering studies that involve subjects solving relatively small tasks. We have focused on small problems, which have been run on only a small number of processors. The code profile of a small problem is qualitatively different form the code profile of “real life” computational science codes (e.g. much more of the code will be devoted to issues other than the core parallel algorithms that are the focus of smaller problems). Therefore, any conclusions that relate to lines of code (e.g. total size, effort per line of code) may not generalize to larger codes in their entirety. In terms of performance, achieving good speedup on 8 processors is quite different from achieving performance on 800 processors. This work does not address the issues that arise when trying to scale to a large number of processors.

6 Future Work

There have been some questions about whether graduate students in parallel programming classes are representative of typical HPC programmers. We plan on observing expert HPC programmers as they solve the same problems that were involved in this assignment. This would give us some indication of how different graduate students are from experts.

Through these studies, we have learned much about capturing programmer effort throughout development of parallel programs. For the next stage of our research, we plan on studying programming problems on a larger scale than classroom assignments. We will be conducting case studies on parallel programming projects in government research labs (e.g. porting a parallel code from one architecture to another).

We also plan on leveraging the knowledge of the HPC community on parallel programming issues by conducting surveys about HPC folklore and defects related to parallel programming.

7 Conclusions

We have so far been successful at forging collaborations among researchers in software engineering as well as high-performance computing, as well as adapting an empirical approach to the study of how HPC codes are engineered effectively.

Our analyses have indicated a number of lessons learned for measurement in this area, including:

- SLOC alone is not a good proxy for effort, especially for comparisons across different HPC approaches. Our data showed that a typical line of code has an associated cost that will vary from one approach to another.
- Code expansion rates highlight differences in the strategies imposed by different HPC approaches. For example, there is a statistically significant difference between the code expansion rates for MPI and OpenMP, which reflects differences in the programming strategies employed in each of those approaches.

These lessons will be useful for contributing to a more comprehensive view of the time to solution metric, taking into account both the development and performance time.

Furthermore, the studies performed to date offer experimental design templates, data collection mechanisms, and baseline data about how novices perform on various HPC applications, which can be useful for both researchers and educators who would like to replicate those experiences. Such replications are crucial if we are to build a true body of knowledge that can more accurately reflect the expected contributions of various HPC practices in different contexts.

This work represents a beginning in exploring the influence of context variables and understanding their effects on high performance technology. Such work will be guided by the matrix we are building to describe the problem space (Table 1) to show where we are lacking coverage. Future studies will be run to generate the large and diverse data sets, covering a majority of cells in the matrix, that will be required to address those questions in the most general case. We are also beginning to work with industrial and government HPC developers, to determine in what ways experience level affects development practices and results.

8 Acknowledgements

This research was supported in part by Department of Energy contracts DE-FG02-04ER25633 and DE-CFC02-01ER25489, and NASA Ames Research Center grant NNA04CD05G to the University of Maryland. We wish to acknowledge the contributions of the various faculty members and their students who have participated in the various experiments we have run over the past 2 years. This includes Alan Edelman at MIT, John Gilbert at the University of California Santa Barbara, Mary Hall at the University of Southern California, Alan Snavely at the University of California San Diego, and Uzi Vishkin at the University of Maryland.

9 References

1. S. Asgari, V. R. Basili, J. Carver, L. Hochstein, J. K. Hollingsworth, F. Shull, and M. V. Zelkowitz, "Challenges in Measuring HPCS Learner Productivity in an Age of Ubiquitous Computing: The HPCS Program," *In Proceedings of ICSE Workshop on High Productivity Computing*. May 2004, Edinburgh, Scotland, pp. pp. 27-31.
2. J. C. Browne, T. Lee, and J. Werth, "Experimental Evaluation of a Reusability-Oriented Parallel Programming Environ-ment," *IEEE Transactions on Software Engineering*, **16**(2), 1990, pp. 111-120.
3. D. Campbell and J. Stanley, *Experimental and Quasi-Experimental Designs for Research*. 1963, Boston: Houghton Mifflin.
4. F. Cantonnet, Y. Yao, M. Zahran, and T. El-Ghazawi, "Productivity Analysis of the UPC Language," *IPDPS 2004 PMEO workshop*. April 2004, Santa FE, NM.
5. J. Carver, L. Jaccheri, S. Morasca, and F. Shull, "Issues Using Students in Empirical Studies in Software Engi-

neering Education.," *Proceedings of 2003 International Symposium on Software Metrics (METRICS)*. Sep., 2003, pp. 239-249.

6. B. L. Chamberlain, S. J. Dietz, and L. Snyder, "A comparative study of the NAS MG benchmark across parallel languages and architectures," *SC'2000*. Nov. 2000.
7. D. D. Howell, *Statistical Methods for Psychology*. 5th ed. 2002, Pacific Grove: Duxbury.
8. L. Hochstein, V. Basili, M. Zelkowitz, and J. Carver, "Combining self-reported and automatic data to improve effort measuremet," *(Under submission)*.
9. A. Hoisie, O. Lubeck, H. Wasserman, F. Petrini, and H. Alme, "A General Predictive Performance Model for Wavefront Algorithms on Clusters of SMPs," *Proc. ICPP*. 2000, pp. 219-229.
10. P. Husbands and C. Isbell, "MATLAB*P: A tool for interactive supercomputing," *Proceedings of The Ninth SIAM Conference on Parallel Processing for Scientific Computing*.
11. A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha, "A Framework for Application Performance Modeling and Prediction," *Proceedings of SC2002*. Nov. 2002, IEEE.
12. D. Szafron and J. Schaeffer, "An Experiment to Measure the Usability of Parallel Programming Systems," *Concurrency: Practice and Experience*, **8**(2), 1996, pp. 147-166.
13. U. Vishkin, S. Dascal, E. Berkovich, and J. Nuzman, "Explicit Multi-Threading (XMT) Bridging Models for Instruction Parallelism," *Proc. 10th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*.