Studying Code Development for High Performance Computing: The HPCS Program

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

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

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

Abstract

The ability to write programs that execute efficiently on modern parallel computers has not been fully studied. In this DARPA-sponsored project, we are looking at measuring the development time for programs written for high performance computers (HPC). Our goal is to measure such development time in both student programming (initially), and then later with professional expert programmers. This paper describes the overall goals of the program and our progress to date.

1. Introduction

The development of High-Performance Computing (HPC) programs (codes) is crucial to progress in many fields of scientific endeavor. However, HPC machines are difficult to program. Effective programmers are rare because HPC code development requires individuals who are both experts in the HPC architecture and in the application domain. These problems will only increase in the future as tougher problems are attacked and more powerful (yet likely more difficult to program) HPC machines are created.

Current activity is mostly focused on better execution performance of HPC codes. However, to avoid potential problems in the future, insight is needed into the process by which codes are created in the first place. We need to understand how effective development of HPC codes currently happens, and where the problems or bottlenecks are. This would not only allow future research into improving development of the high-payoff problems and provide the most useful support, but may also improve our knowledge of good practices for HPC development that can be passed along to novices.

In order to investigate these questions, DARPA has funded the High Productivity Computing Systems (HPCS) project. This project is a collaboration among researchers experienced in empirical studies of software engineering (i.e., the work practices required for production of quality software so far studied in non-HPC domains) and researchers in the area of HPC itself.

In this paper we provide an overview of the HPCS project, its research goals and areas of concentration. We describe our early work so far and initial lessons learned, and discuss how we are scaling up our improved research protocols to attack the research questions in later work.

2. Research Goals

The goal of this work is to better understand and quantify software development for high performance computers. Currently there is little empirical evidence to support many of the implicit assumptions made by the HPC community. This work will help to determine whether those assumptions are true and provide better guidance for planning and decision-making in HPC code development. Our work will begin with an observational approach, observing HPC code development by students and professionals to understand basic practices, to generate a series of well-grounded hypotheses and validate them where possible. As our understanding of basic phenomena increases the research is expected to move into a mode of testing key hypotheses.

We have identified the following focus areas, representing important phenomena in HPC development:

- **Tradeoff between development and execution time.** In developing HPC software, *time to solution* is an important metric. For many applications, the value of a result declines if it cannot be obtained by a deadline. 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.

Currently in the HPC community, human activity is often not empirically measured as rigorously as execution time. Both development time and execution time play crucial roles in the overall time to solution, so we believe that empirically measuring development time is important.

The overall goal in HPC development is to reduce the time to solution, so researchers can strive to reduce either the development time, the execution time or both. 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. Often a solution is programmed to execute on a single processor (e.g., a serial solution), and then the program is tuned to operate more efficiently on a multiprocessor HPC system. 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.

By studying the development and execution times associated with implementing various codes using different development approaches like OpenMP [4] or MPI [3], the tradeoffs inherent in those approaches can be better understood. This information will assist in planning so as to achieve optimal values for development time and execution time, such that time to solution is minimized. Such decision-making will obviously vary based on the circumstances of use for the software. If the code will be executed many times, then the cost of increased development time can be amortized across multiple runs of the software and balanced against the cumulative reduced execution time. Conversely, if the code will only be used once, the benefit of increased effort tuning the code may not be as large.

- **Workflows.** Studies performed by HPCS will also examine development workflows (i.e. the series of activities performed by a developer working on an HPC code), looking for common patterns that can be said to characterize how people go about developing HPC codes. We anticipate this information will be useful for a variety of reasons:
  - To understand if there are different working styles, i.e. different sets of strategic rather than tactical choices that can be made about how to go about development, and whether different styles have better correlations with cost-effective and successful development.
  - To understand where most of a developer's time will be spent, so that research efforts aimed at improving the development process can be best focused for overall impact.
  - To give better guidance to novices learning how to develop codes for the first time.
- **Developer skill/experience.** Another important issue that needs to be studied is the level of expertise a developer needs in order to effectively create high performance software. If there are only a few experts in the world who can effectively develop HPC solutions, then the number of problems that can be solved is greatly limited. Conversely, if novices can efficiently develop effective HPC software, many more problems can be addressed because of the larger base of developers. There is the assumption that the solutions produced by novices will not execute as fast as solutions produced by experts and may take slightly longer to build. One of the goals of this project is to begin characterizing the abilities of novice developers in relation to expert developers.

3. Work to date

In order to begin answering the research questions posed in the previous section, a series of empirical studies has been planned. This type of empirical research is novel for the HPC community, however, so prior to conducting rigorous (but expensive) full fractional factorial experiments with professionals, it was necessary to conduct pilot studies to debug the experimental methods and techniques. Our first research activity was running a pre-pilot study aimed at understanding the issues involved and debugging our methods.

The setting for the pre-pilot study was a graduate level High Performance Computing class taught by Jeffrey K. Hollingsworth at the University of Maryland. The students in this class were taught the concepts associated with HPC, so it was an ideal place to begin evaluating the performance of novice HPC developers. For most of the students in the class, it was their first time developing a HPC software application.

Fifteen students took part in the study, which consisted of two assignments. In the first assignment, students were to create a sequential and then a parallel solution using MPI for the "Game of Life" problem [2]. Subjects were graded on, and hence had incentive to focus on, the correct performance of both sequential and parallel versions and the amount of speed-up achieved for the parallel version. In a second assignment, subjects switched to OpenMP and had to devise a parallel version of the SWIM benchmark.

In both assignments, students were required to manually report on their total daily effort and the sequence of tasks on which they spent their time. The compiler and job scheduler were also instrumented to collect some of the same information automatically.

The results of this pre-pilot study are a series of lessons learned that allow us to design more effective pilot studies. Another result of this study is a set of initial hypotheses that can be further refined in the pilot studies and tested during later full experiments.

Lesson 1 – *Separate the serial coding and parallel coding into two assignments*

In this study, the students were only asked to submit the final (parallel) version of their code. While we did capture intermediate versions via the compiler

instrumentation, we could not definitively determine which version was the final serial version. There were two types of desirable data analyses that could not be accurately performed because of the lack of this separation of serial activities and parallel activities.

First, the execution time (performance) of a subject's serial code needed to be compared to the execution time of his or her parallel code run on various numbers of processors. This analysis is used to determine the amount of speedup achieved by the subject. Because we did not have a serial version of the code, we had to approximate this metric by using the performance number of the parallel code run on only 1 node.

Second, in our analysis, we often wished to separate out the effort expended during serial coding from the effort expended during parallel coding. The manually reported effort data, which did separate the serial and parallel activities, was not very reliable (see Lesson 3 below). So, in order to have an accurate separation, we needed to be able to separate the effort captured via the compiler instrumentation into serial effort and parallel effort. Because the serial code was not submitted, giving us a definitive end date for serial coding, we had to develop an algorithm to approximate the point at which serial coding stopped and parallel coding began.

For future studies, we suggest splitting the coding assignments into parts. In the first part, the subjects are instructed to solve the problem by writing a serial program. Once the serial program is completed and submitted, then the subjects can begin working on parallelizing the serial code already created.

Lesson 2 *– Account for uncollected data when subjects work on uninstrumented machines*

As we began to analyze the automatically collected data, it became obvious that many of the subjects did some of their work on machines that were not instrumented to collect data automatically. In hindsight this occurrence is not surprising but it is something that was not accounted for during the planning and design of the study. The automatically collected data indicated that many of the subjects did not begin working on the instrumented machines until they needed either the MPI compiler or the use of multiple processors to test their parallel code. To make matters more complex, an MPI version of the C compiler is standard on most Linux implementations, so a student with access to a Linux machine could effectively finish the project before submitting a final run on the instrumented HPC system. This observation means that the automatically collected data was not collected for much of the serial development step and potentially for the parallel tuning effort.

There are two possible solutions to this problem for future studies. First we can ask the subjects to work only on the instrumented machines, thereby allowing us to automatically collect data for all of their development work. Secondly, we can develop a small script that subjects can install on any other machine on which they work that will collect the same data as the script on the main machine. Neither of these solutions is ideal, so we are continuing to pursue other solutions to this problem.

Lesson 3 *– Manually reported data is suspect.*

One of our primary objectives in the pre-pilot study was to understand the quality of the data we collected using our initial mechanisms. Therefore we spent some time analyzing the correlation between our sources of data. The details of this are discussed more fully in a companion paper [1]. We only summarize the results here.

We treat the automatically collected data as more accurate, since this data was objective (not reliant upon subjective reporting by humans), unobtrusive (not interfering with normal work processes) and automatable (not dependent upon active reporting by human). Subjects were aware they were being monitored, but not aware of what was being observed or why. This included not only the log of compilation and execution activities, but also a database that was created containing captured source code and test data used throughout the development process.

An approximation of effort was done based on the number of 'job steps,' which were defined as actions that directly reflected the frequency of computer system use (i.e. compilations and test executions). An assumption was made that each of these activities involves a certain expenditure of a programmer's effort, so they indirectly quantify the effort not related to thinking or design activities.

However, we were not able to find any significant correlation between the effort data extrapolated from the subjects' automatically collected data and the manually reported effort. We were thus forced to treat the manual data more skeptically. Hence,

Lesson 4 *– Data collection and analysis should be as automated as possible*

Of course, a central weakness of automated collection is that while the data can tell us what was done on the computer, it doesn't provide information about how those activities contribute to the decision making process in code development. A key research goal is to increase the usefulness of the data collected from automated mechanisms without making it more obtrusive to the developer.

Based on these observations, a series of hypotheses has been formulated for more specific investigation in future studies. These hypotheses include items such as:

- Workflows will be different for developers with less programming experience than for developers with more programming experience
- The effort required to produce a parallel solution to a problem is greater than the effort required to produce a serial code to solve the same problem.

- There is a large variation in the overall amount of effort among developers, but the distribution among the various activities is similar.
- For a specific problem, the mean performance of MPI programs will be higher than the mean performance of OpenMP programs.

4. Ongoing Work

Based on the lessons learned from the pre-pilot study, during the Spring 2004 semester we had a number of pilot studies ongoing. The pilots are exploring a number of different research questions, using different protocols. All will be able to feed suggestions for improvements back as we understand the results. All of these studies will be complete by the time of the workshop at ICSE, so we will be able to discuss initial results by that time.

The pilots are being done in classroom environments, with the following instructors:

- Mary Hall (University of Southern California): 10 subjects are given a small serial program (LU decomposition) and asked to tune it to improve performance, then produce a parallel version of the same solution using MPI, and tune that.
- Uzi Vishkin (University of Maryland): This is a class on parallel algorithms. 17 subjects are asked to produce solutions to two different assignments (array compaction, randomized selection) directly in a parallel environment, without first producing a serial version.
- Alan Edelman (MIT): Subjects are asked to produce versions of the same code (for Buffon-Laplace Needle Problem, grid of resistors, Laplace's equation) using MPI, OpenMP, and MATLAB *P.
- John Gilbert (University of California Santa Barbara): 43 subjects are given the same assignment using multiple parallel approaches, as in the study with Alan Edelman above.
- Alan Sussman (University of Maryland): Subjects are asked to solve the Game of Life using MPI (replicating the first part of our pre-pilot study).

Although there are many differences among the studies, even in terms of the specific hypotheses being investigated, they all share the same general framework in terms of dependent and independent variables:

- Independent (predictor) variables:
  - HPC approach (e.g. OpenMP, MPI, etc.)
  - Types of subjects (level of previous expertise, skill, workflows used, etc.)
  - Application/problem for which an HPC solution is being designed
- Dependent variables:
  - HPC development effort
  - HPC development effectiveness (measured for example by the quality of the resulting HPC code)

As a result, this set of studies will result in data well suited to exploring the effects of different HPC development approaches on different problem types. Some data will come from the same subjects performing different types of tasks; others will reflect the same task addressed by subjects in different environments and with different backgrounds and skill levels. These data sets will form the basis of future data needed to explore the relationships among our phenomena of interest.

Other lessons learned so far are mainly procedural, for example, understanding how much time needs to be included in the schedule for receiving approval from the various educational institutions for performing studies on human subjects.

5. Future Work

The ultimate goal of this work is to run full fractional factorial experiments with HPC code development professionals, to investigate specific hypotheses resulting from our earlier pilot studies with the most rigor. In such an experiment, we envision that subjects will use two or more parallel programming approaches to implement different benchmark applications. The order of the approaches and benchmarks can be varied to combat the effects of subjects learning from one assignment to the next. Such an experiment will help us to better quantify the tradeoffs between the different approaches for different types of benchmarks.

To do that, we will be able to reuse the refined instrumentation and our experience with empirical study designs and HPC environment data collection mechanisms, which we have been experimenting with in the meantime.

The end result of such studies will be well-formulated and tested heuristics concerning the aspects of human developers, HPC architectures, and code development practices that work together to influence the time to solution of problems being tackled using HPC approaches. That knowledge, in turn, is necessary to be able to plan and meet the current and increasing challenges in a number of important scientific fields.

6. Acknowledgements

This work is sponsored by the DARPA High Productivity Computing Systems program.

7. References

[1] S. Asgari, et al, Challenges in Measuring HPCS Learner Productivity in an Age of Ubiquitous Computing (submitted to this workshop)

[2] M. Gardener, MATHEMATICAL GAMES: The fantastic combinations of John Conway's new solitaire game "life," Scientific American 223 (October 1970): 120-123.

[3] Message Passing Interface Forum, http://www-unix.mcs.anl.gov/mpi/mpi-standard/mpi-report-2.0-sf/mpi2-report.htm

[4] OpenMP Architecture Review Board, OpenMP C and C++ Application Program Interface, Version 2.0, March 2002.