Challenges in Measuring HPCS Learner Productivity in an Age of Ubiquitous Computing

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

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

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

Abstract

Collecting development data automatically is difficult in this era of ubiquitous home computing. This paper describes our efforts in the High Productivity Computing Systems project to better calculate effort data among a set of student programming exercises.

1. Introduction

As in other types of software development, the usual goal of developing codes in High Performance Computing (HPC) is to arrive at the solution of a problem with minimal effort and time. Thus, an important metric for evaluating various approaches to code development in HPC is "time to solution," encompassing both the effort required to understand and develop a solution as well as the amount of computer time it takes to execute that solution and arrive at an answer.

Metrics and even predictive models have already been developed for measuring the code performance part of that equation, under various constraints (e.g. [4, 8]). However, little empirical work has been done to date to study the human effort required to implement those solutions. 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. Researchers in the HPC community associated with DARPA's High Productivity Computing System (HPCS) project[1] have decided that it is important to begin to understand empirically whether or not the general assumptions that are guiding decision-making are true.

As a first step in this direction the members of HPCS are executing a series of empirical studies. The overall goal is to study the human effort required to develop solutions to various problems using different HPC approaches and languages. As data is collected about the implementation of various solutions, the amount of effort necessary for various applications and various approaches can be characterized. This data will allow heuristics to be developed to decide which approach(es) should be used in a given environment. These heuristics will provide a more rigorous basis for making the decisions that are currently being made without empirical evidence.

[1] http://www.darpa.mil/ipto/programs/hpcs/index.htm

This type of empirical research is novel for the HPC community, so we have begun by conducting some pilot studies to debug the experimental methods and techniques. In this position paper, we briefly describe our efforts to date to adapt mechanisms for measuring developer productivity to the HPC domain, along with aspects of the HPC domain that pose unique – and not yet completely solved - problems.

2. Survey of past approaches

Collecting effort data in this age of ubiquitous personal computers is hard, and working in an HPC-specific environment complicates things even further by imposing some unique constraints. We are finding that methods that have been used in past successful studies of development effort are no longer applicable, at least not "out of the box."

- Method 1: In the 1970s, computers were big and expensive. Many users were forced to use the same machine as it was the only choice available. Collecting data was therefore easy – compilers could be instrumented and would catch and report all computing activity for later analysis (e.g. [1, 9]). Life was easy then.
- Method 2: A later style of data collection is that used in the 1980s and 1990s in NASA's Software Engineering Laboratory. Compilers were not instrumented but subjects reported effort manually through time sheets. Since this was part of the job description and part of training, the resulting data was judged to be reasonably accurate. This was sufficient to allow correlations to be found between effort and other aspects of interest, and several good studies resulted [2].

Our original thought in planning the HPC pilot studies was that the development environment for HPC code development would be similar to that described in Method 1 above (i.e. a single compiler and environment that must be used by all users). However, as described below, this turned out not to be the case. Almost every subject had access to their own computing resources, which were easier to access and use than the parallel cluster used for the course, and some extra effort on their part seemed well worth it in order to use their own resources as much as

possible and port their code to the cluster as late in the process as possible. As it turned out, only final tailoring to the parallel cluster and performance tuning on multiple processors really had to occur on the parallel machine. Even the operating system/development environment, which we expected was rare and expensive enough that subjects would be forced to use the single machine under our control, turns out to be shipped for free with the Linux workstation environment.

3. Work to date

In a pre-pilot study at the University of Maryland, we decided to use both manual and automatic data collection mechanisms, to begin experimenting with mechanisms tailored to HPC code development. In this study, subjects were asked to undertake two programming assignments on a parallel cluster, using two different development approaches: MPI [6] and OpenMP [7]. 15 students in a graduate-level High Performance Computing course participated.

Manual. We developed a series of forms that subjects can use to report their effort and background information. Some key variables we asked for include:

- Educational background (related to HPC development);
- Native language;
- Prior evelopment experience (overall software experience as well as parallel-specific experience);
- Problem domain experience.

Perhaps most importantly, we created a log form that subjects are asked to use to keep track of the effort spent on the project over time and the various tasks they performed with that effort:

- Thinking/planning
- Coding a serial implementation/Reading and understanding the serial code
- Parallelizing the serial implementation
- Tuning the parallel code
- Testing the code
- Other

Since the pre-pilot, we have developed tools so that the data collection is now entirely web-based, and the effort for the user is minimized.

Automatic. To have a more objective way to collect data about effort and activities, we created a wrapper (consisting of a C program and two Python scripts) for both the MPI C compiler and the job submission program. When either the compiler or the job submission program is invoked, the wrapper logs a timestamp, the user's name, and any flags sent, before passing execution to the intended program. Additionally, when the compiler is invoked the wrapper logs the entire source file, and the user must choose the reason for compilation from a short menu consisting of:

1. Adding functionality (serial code)
2. Parallelizing code
3. Improving performance (tuning)
4. Debugging: Compile-time error on previous compile
5. Debugging: Crashed on previous run (segmentation fault)
6. Debugging: Hung on previous run (deadlock, infinite loop, etc.)
7. Debugging: Incorrect behavior on previous run (logic error)
8. Restructuring/cleanup (no change in behavior or performance)
9. Other

The reason chosen is stored along with the other information captured for that compile. Post-hoc questionnaires and interviews with subjects confirmed most subjects did not perceive the instrumentation as notably onerous.

Aside from being asked to choose the reason for compilation, the behavior of the wrapped programs is indistinguishable to the user from their normal operation.

We are currently experimenting with ways to incorporate the automatic collection tools into a package that will be available for other researchers to use with minimal tailoring required.

4. Observations

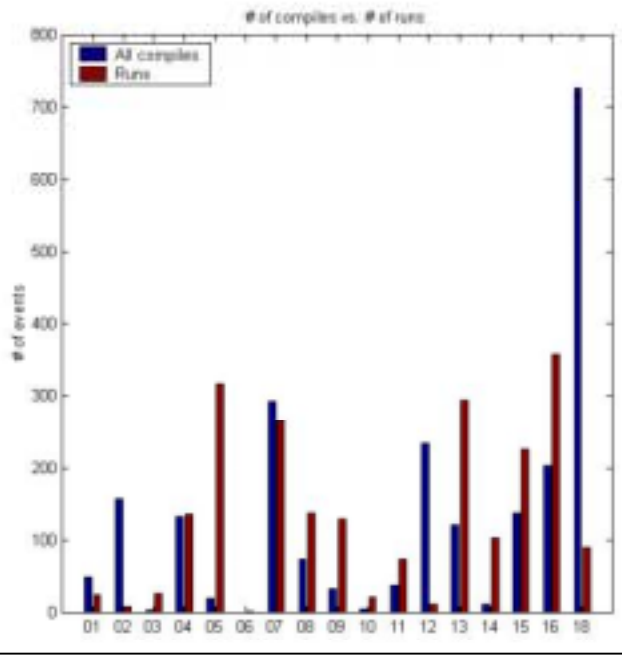
The automatically- and manually-collected data provided different kinds of insight since the data describe different levels of granularity. However, although we spent a significant amount of time on making sure the data mechanisms were easy to use and as unobtrusive as possible, there are some seeming anomalies in the data that require further investigation.

Summary of automatically-collected data. We summarize the data collected via the compiler and job submit instrumentation for each subject in Figure 1. Although we had at first expected to see a rough parity between the number of compiles and the number of times the code was run, the data show that there is not necessarily any such clear relationship. In the current study we were not able to explore the reasons for this, but some initial hypotheses do exist.

A larger number of runs than compiles may indicate:

- Subjects exhaustively tested their code at various points during development, on multiple data sets, perhaps as part of performance tuning.
- A significant amount of development was done off the cluster, and the cluster was used mainly for accurately measuring code performance.

Figure 1



- Subjects had difficulty with the syntax of the job scheduler and repeatedly sent jobs that immediately came back as errors.

A larger number of compiles than runs may indicate:

- Subjects were "thrashing," i.e. were trying to develop the code quickly to turn in the assignment rather than optimizing performance or correctness of output.
- Subjects spent an inordinate amount of time on debugging, responding to compiler errors.

Generating workflow descriptions from automatically-generated data. The timestamp data allowed us to understand the chronological series of events and look for various workflow patterns in how subjects attacked the problem. Specifically, we wanted to see the relation between the effort spent on serial versus parallel coding, and on functional development versus performance tuning. To do this, we mapped the data recorded in the log (especially focusing on the "reason for compilation," whose possible values were described in Section 3) to a smaller set of activity types: If the user explicitly gave "serial", "parallel", "tuning", restructuring", or "other" as the reason for compiling, then that was simply used as the activity category. Runs were classified as "testing". If the user was debugging, then the event was classified based on the previous event (e.g. if the previous event had been serial, then the debugging was classified as serial work, if the previous event had been parallel, then the debugging was classified as parallel work, etc.).

The data does show some high-level patterns. For example, Figures 2 and 3 show two different styles of iteration through the key tasks of adding serial functionality, adding parallel code, testing, and performance tuning.

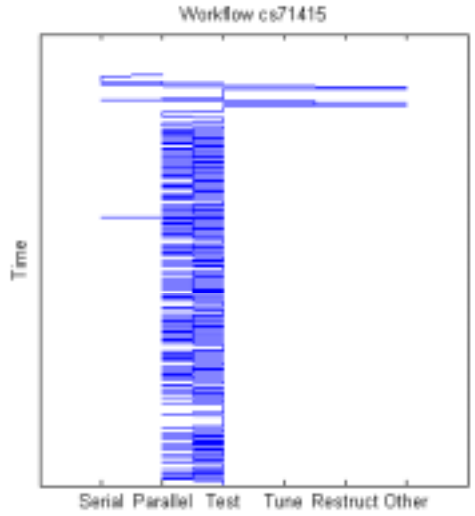
Summary of manually-collected data. We used the manually-completed time and activity logs to investigate a more full picture of development effort, including time spent off of the computer. Results are shown in Figure 4. Most interestingly, although the total effort reported by subjects through the manual logs varied widely in its absolute value, the relative distribution among the activities was similar across all of the subjects.

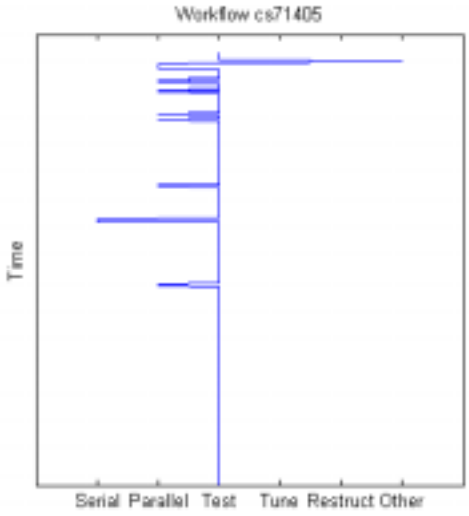
Correlating automatic and manual data. To validate the accuracy of the data, we tried to correlate the results from the two methods. Unfortunately, in doing so we find wide discrepancies. The correlation was done initially by making estimates about the total effort spent by subjects based upon the timestamps recorded in the automatically-generated logs. For each subject, the time between any two events (either compiles or runs) in the log was calculated. If the time interval was less than a specific threshold (in this analysis we used 45 minutes), that interval was added to the subject's effort total. As shown in Figure 5, no correlation between this estimate and the manually-reported data is detectable.

Furthermore, no such correlation was detected even after we accounted for the fact that significant amounts of work might have been done off of the instrumented cluster. To make the estimate more accurate, emails were sent to students after the experiment asking them to estimate what percentage of their development effort had been spent on the instrumented machine. Based on these percentages, the instrumented effort was adjusted, but there was still no correlation detected with the manually-reported effort.

Puzzled by this discrepancy, we investigated whether the *days on which effort was spent*, reported in the manual data, matched the days recorded in the timestamp logs. We found several discrepancies, which were not consistently associated with particular subjects and which did not have a consistent duration. Also, there were no obvious "holes" in the timestamp logs when no data was recorded for any subject. The only remaining explanation seems to be that subjects were simply inconsistent in their effort reporting.

Figures 2 and 3





New Hypotheses. We viewed this study as a chance to generate well-grounded hypotheses in a new area, since we had no formal hypotheses prior suitable for testing. The data described in this paper is just a sampling, but has allowed us to generate hypotheses such as the following for future testing:

- There are four workflows for parallel programming:
  - d_1: develop and test in small increments,
  - d_2: develop in small increments with a long sequence of tests after that,
  - d_3: develop in large chunks and test after each large development,
  - d_4: develop in large chunks with a long sequence of tests after each large development
- There is a large variation in the overall amount of effort among developers, but the distribution among the various activities is similar

5. Future Work

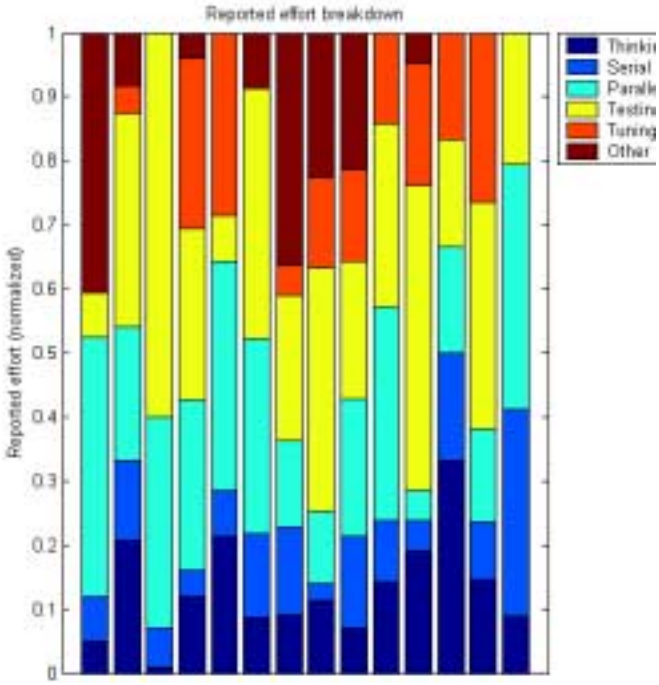
Our analysis has indicated several difficulties in measuring learner effort in HPC tasks:

- The need to capture effort and activities from across several machines, some of which may be out of the experimenters' control.
- The inherent unreliability of manually-reported data.
- The need to make inferences (based on guesses, heuristics, and rules of thumb) to map automatically-collected data to our real measures of interest, in the absence of an accurate source of data for verification.

Our future work will focus on overcoming these challenges. One possibility will be to investigate whether we can develop mechanisms for better process conformance to the data collection procedures (for example, by not letting subjects submit their program until all previous data has been submitted).

One solution we are exploring is to analyze the activity data in greater detail, incorporating assumptions about chronological order in order to make better estimates about the task being undertaken. For example, a lot of compiles in rapid succession would suggest debugging, while alternating between compile and execution or multiple executions in quick succession might suggest testing of the code. More ambitiously, if we can pinpoint

Figure 4



the differences between successive versions of the code, we can develop heuristics about the activity that was ongoing in that time period. For example, if the delta contains no editing on statements involving parallel operations, then we can infer that the subject was doing serial coding.

It may also be the case that we simply need to collect more or different data. Philip Johnson's tool HackyStat [5] is one possible answer we are exploring. It can be tailored to work with a number of different editors, and reports the amount of time an editor is "live," providing a better baseline of overall effort. However, we haven't found a way to cross-index this with specific tasks yet (e.g. to know when a subject is parallelizing vs. tuning code). We are also considering the use of an extensible IDE, like Eclipse [3], that would allow us to collect more accurate data.

In further research, we will broaden our focus to industrial and government HPC environments to understand how to tailor these methods to for use by professional subjects.

6. Acknowledgements

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

7. References

[1] V. R. Basili and A. J. Turner, Experiences with A Simple Structured Programming Language, in Proceedings of the Fourth Symposium on Computer Science Education, ACM 1974; SIGCSE Bulletin, February 1974.

[2] V. R. Basili, M. Zelkowitz, F. McGarry, J. Page, S. Waligora and R. Pajerski, Special Report: SEL's Software Process-Improvement Program, IEEE Software, Volume 12, Number 6, pp 83-87, November 1995.

[3] Eclipse.org. http://www.eclipse.org/

[4] A. Hoisie, O. Lubeck et al., "A General Predictive Performance Model for Wavefront Algorithms on Clusters of SMPs," Proc. ICPP 2000, 219-229.

[5] P. M. Johnson. Hackystat system. http://csdl.ics.hawaii.edu/Research/Hackystat/.

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

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

[8] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia A. Purkayastha, "A Framework for Application Performance Modeling and Prediction," *Proceedings of SC2002*, IEEE, Nov. 2002.

[9] M. Zelkowitz, Automatic program analysis and evaluation, Second International Conf. on Software Engineering, San Francisco, CA (October, 1976) 158-163.

Figure 5