Climbing the Plateau

Getting from Study Design to Data that Means Something

Christine A. Halverson
Social Computing Group
IBM T.J. Watson Research Center
krys@us.ibm.com

Jeffrey Carver
Department of Computer Science
University of Alabama
carver@cs.ua.edu

Abstract

Evaluating the usability of a programming language or tool requires a number of pieces to fall into place. We raise issues along the path from study design to implementation and analysis drawn from the experience of running several studies concerned with a new parallel programming language – X10. We summarize several analyses that can be drawn from different aspects of the same data.

Categories and Subject Descriptors D.2.5 [**Software Engineering**]: Testing and Debugging; H.1.2 **[Information Systems]**: User/Machine Systems—Human Factors

General Terms Measurement, Experimentation, Human Factors, Languages.

Keywords parallel programming; programmer productivity; study design; data collection; integrated methodology; analysis of programmer behavior.

1. Introduction

While programming massively parallel computer systems has been going on for at least two decades, few details are known about its practice. The information that is available is a combination of anecdotal evidence and a few studies of parallel programming students. Neither of these sources is bad, just limited.

In contrast, scientific computing "in the wild" using high performance computing systems covers a wide range of problems within a variety of organizational structures: industry, academic, public and clandestine government. While scientific software has a lot of issues in common with "traditional" software, it also has many unique issues due to its use of massively parallel machines and the fact that the primary job and skill set of many of the programmers is science (or other work) not computer science.

This work was motivated by our participation in the DARPA High Productivity Computing Systems (HPCS) program [10]. The program is in the last of three phases of an eight-year effort aimed at developing peta-scale machines that significantly improve the productivity of its users, i.e. scientists, programmers, data managers and system administrators.

As a member of the PERCS Productivity team at IBM [15] the first author is working to demonstrate an increase in programmer productivity with IBM's new machine and tools from a 2002 baseline. While definitions of productivity are contentious, two things are clear. The ability of programmers to use the machine and its software stack (including programming language and assorted tools) – that is, the *usability* of all of these things – is important. We also must compare usability between 2002 and what will be available around 2010.

We need to understand how parallel programming is occurring and how its practice is impacted by differences in machines, tools and languages. In this paper we outline the path from study design through study implementation and analysis for just one of the many cases we need to evaluate. In particular we focus on how choices made early in study design impact the kinds of analysis that can be done.

2. Related Work

There is literature from a number of disciplines that influenced our approach to study design. Previous efforts to understand programmer behavior have encompassed three main methodologies: self-report via survey or interview, automated measurement of machine-human interaction during programming, and empirical studies – whether in the laboratory or in the field. (For example. See Perry et al. [16,17] using field based approaches; Hofer and Tichy [9]

PLATEAU'09 Workshop at Onward! October 26, 2009, Orlando, Florida, USA.

for a review of empirical approaches in the last decade; and Basili [2] for lab based empirical approaches.)

There have been studies in HPCS since the early 80's. Some studies centered on specific machines, while others focused on programming languages or parallel environments [3]. The manner of data collection and the available subject pool have largely constrained the method and measurements. As most studies occurred in universities, the subjects were students, usually in their first parallel programming class [7]. The programming task was generally a class assignment, such as Sharks and Fishes [8]. Course requirements also dictated the programming language. Tools and machines were determined respectively by personal preference and availability. As in earlier studies on sequential programming, data collection is one of three types: manual, automated or *hybrid*, a combination of the two. Hochstein et al [8] present a *hybrid* method that combined automated data gathering with manual data provided by programmer self report.

We have used a different combination for a hybrid–the integrated methodology [4]. This method combines automatic data collection (SUMS, [13,14]) with concurrent manual observations. These observations are taken by a trained independent observer, eliminating two of the problems noted with self-report data, namely the self-interruption requiring context switching by the programmer and the potential bias expressed in the content of the programmer's self reports.

Methodological difference aside, one of the pervasive problems in empirical studies of programmers is the ecological validity of the study. Ecological validity refers to how close the method, setting, and materials mimic the real world situation. Controlled experiments by nature limit the variance in what is being studied. The recognition of the necessity for ecological validity in programmer studies is not new. Schneiderman and Carroll [18] focused on the need for studies of professional programmers in their native environments, and Perry et al picked up this call again in the mid 1990s [16,17]. In both cases researchers were figuring out ways to study programmers in the wild, which resulted in qualitative and quantitative data.

3. Study Design: Building the Trail

Designing the study is laying out the trail to be traversed. The point is to consider all the problems and issues in advance so that during the study the right type of high quality data can be collected.

Our overall goals were to provide a baseline (circa 2002) for single programmer behavior and show progress in productivity improvement delivered by the X10 programming language [21] and the X10 development toolkit (X10DT). We made the following choices in study design.

SUBJECTS: Prior studies were mostly done with students who were taking their first parallel programming classes [7,8]. For PERCS, we recruited students with programming experience and taught them one of 3 parallel languages [4,5]. (See also [19] re designing studies for HPCS). However, feedback from the HPCS program noted that the results of these studies didn't necessarily apply to actual experience levels.

In response, we chose to look at two levels of experience. Prior trouble recruiting HPC professionals led us to define experienced subjects as those having 10 or more years of experience in parallel programming without constraining where they worked currently. To avoid some student issues novices were considered to be those that have had at least one parallel programming class and at least 3 years of experience programming. In this way we hoped to bracket the problem space and avoid effects caused by having just learned parallel programming. Subjects were also required to be familiar with the most commonly used editors and tools. For the baseline these tools were vim, emacs, and gnu debugging and TotalView. For the newer condition the tools were Java and Eclipse (as the X10DT is built into Eclipse as a plug in).

BASIC DESIGN: To cover both goals and experience levels we ran 4 groups of 10 subjects in a standard 2 x 2 design. This would give us a group of novices and experts for each language condition: MPI and X10. We hoped to target the majority of experienced subjects through our association with National Energy Research Scientific Computing Center (NERSC) and Lawrence Berkeley Laboratories (LBL) while picking up our novices either through post-docs at the same institutions or advanced graduate students associated with them.

PROBLEM: We needed to use a programming "problem" that would be realistic but actually solvable in the time available. In our previous study we used a Synthetic Scalable Compact Application (SSCA). (Developed for HPCS [1]). SSCA1 presents a pattern matching problem, such as gene sequencing. Although it's a problem from genetics, it does not require deep domain knowledge to solve.

From prior experience we knew that the problem was solvable by most within a 2-day time frame. We provide subjects with working serial code and ask them to parallelize a portion. Making it parallel can be done in two ways: a more difficult wave-front algorithm or a straightforward embarrassingly parallel solution. Again, to reduce time, we attempted to focus the subjects on the easier approach.

ENVIRONMENT: There are multiple issues regarding hardware and software setup. We must duplicate 2002 circumstances for the baseline while also accommodating the newer tools. Finally both setups must allow data collection.

For the baseline, our model was an existing machine: NERSC's IBM SP RS/6000 Power3—Seaborg. It was brought online in 2001 and still had the (updated) software

and tools that fit our needs. Our pilot subject used Seaborg until it was decommissioned in January 2008. At that time we switched our setup to Bassi—an IBM p575 Power5 system. While the computational capabilities are somewhat different, we were able to provide the same software stack (operating system, editors, mpi library and compile commands) as we had on Seaborg to approximate 2002 conditions. All study subjects all used Bassi.

Most programmers in 2002 programmed directly on the interactive portion of a machine's nodes via secure shell connection from a desktop. However, others developed code on a local machine and then uploaded and ran the code. We wanted to accommodate both styles. For our purposes, laptops are as powerful as a desktop and more portable. We set up five identical ThinkPad T61p laptops for this study. The table below shows the machine configuration and data collection software used.

	Laptop	Bassi
OS	Fedora Core 6 Linux	IBM POE, AIX
Editors	Vim, emacs	Vim, emacs
Shell	Bash	Bash
Languages	Fortran 77 & 90, C	Fortran 77 & 90, C
Message Passing	MPI	MPI
Web Browser	Firefox (NERSC and language sources)	none
Automated Data Collection	Hackystat Slogger (web) pFig (Eclipse) Istanbul (screen cap)	Hackystat

DATA COLLECTION: We strove for unobtrusive and automated data collection. The first goal is to minimize any effect on programmer behavior while the second goal is to minimize experimenter effort. This means using the computer to automatically collect interaction data. However programmers engage in activities besides typing on a keyboard, so we also needed to collect behavioral data.

For this project we used the Hackystat v7 framework for automated data collection. Hackystat [6] is an open source framework for automated collection and analysis of software engineering process and production metrics. Hackystat users attach software "sensors" to their development tools, which unobtrusively collect and send raw data about development to a Hackystat web server for display and analysis. On both the laptops and Bassi we used sensors attached to the command line, the bash shell, and the vim and emacs editors. On the laptop we also used another piece of software – slogger [20] – to record web browser activity. (Note: subjects were restricted to references for language and machine operation. No googling was allowed.)

The addition of X10 and the X10DT required using Eclipse. The Hackystat Eclipse sensor, like its other editor sensors, captures whether or not a file is being edited and whether the file size is changing. With vim and emacs this information allows us to infer that the programmer has edited and saved the file. Then the user needs to go to the command line to build and run the file. However, as Eclipse is configured to automatically save and build we cannot necessarily make the same inference of programmer intent. Instead we used a separate data gathering Eclipse plug-in written in Java, called PFIG (Programmer Flow Information Gatherer [12]). PFIG instruments Eclipse to monitor navigation activity in the IDE. It collects data such as the location of the text cursor within files, usage of the package explorer, the locations and contents of variables and methods within classes, program launches, and changes to source code. This was more data than we needed, but we did feel that seeing the patterns of use exposed within Eclipse would provide us valuable information about the use of X10 and the X10DT.

Finally, observations were used to fill in gaps where no automated data was collected and to infer programmer intent in some cases. In our previous study [4,5] of 27 subjects, 3 observers could not cover the subjects continuously. In that case, we chose a sampling method where one observer was assigned to a cohort of 9 (one language condition). Observations were taken for 5 minutes on one subject, after which the observer moved to the next subject in the cohort. Unfortunately this meant that each subject was only observed for a 5 minute period out of 45 minutes, which was not always sufficient to cover gaps in the automated data collection. For this study we knew we would have similar limitations on observers. We designed the study so that at any one time each observer only had 3-4 subjects to observe. Each subject was observed at least once per minute insuring coverage with automated data.

We knew there would still be situations where we would have coverage difficulties. Video data would provide similar detail, but while easy to record, it requires fairly obtrusive equipment and faces a scaling problem – 10 subjects require 10 cameras. Our solution was to move to screen capture. While it requires significant time to analyze, this type of data does have the benefit of being relatively unobtrusive and automatically collected. We used an open source project called Istanbul [11] for screen capture. Requiring some additional user setup work it is more intrusive than Hackystat, but the payoff seemed worth it.

Overall we had three kinds of data collected on three different time scales. The Hackystat sensors poll on a 5 -15 second interval. The human observations are within a minute resolution and for the X10 portion the PFIG log was time stamped at millisecond resolution.

4. Running the Study: Climbing Up

Before the study could be run, two additional things needed to happen. One was to get human subjects approval from the appropriate institutional review board (IRB), and

the other is to recruit subjects. IRB approval varies with institution and each situation has its own challenges so we do not deal with it here—other than the caveat to allow plenty of time to deal with this step.

The other issue is actually recruiting, qualifying and scheduling subjects for the study. We changed several aspects of the study to facilitate recruiting We had shortened the study from nearly a week to two days. We also paid subjects and got NERSC management to allow people to participate during work time. We still had trouble getting experienced subjects. Only 4 qualified and participated.

We were more successful with the students at Rice where we recruited subjects for the two novice language conditions. We qualified 9 subjects for the X10 condition and 7 for C+MPI. Due to 2 dropouts we ended up with 7 in each condition.

PROTOCOL: At both NERSC and Rice the set up was the same. The study procedure was close with a few variations. The protocol consisted of:

- Physical setup of machines
- Welcome subjects and obtain consent
- (X10 condition – 1 day language tutorial)
- Cover basics of machine and problem organization
- Introduction to problem
- Coding
- Daily breaks for lunch and snack
- Complete problem
- Take post survey

All subjects worked in the same room, facilitating the process for the experimenters.

Subjects were provided with electronic and hard copies of the problem statement and associated materials about the details of the machine set up. They also had pen and paper for note-taking. Subjects were cautioned not to confer about the problem, but were able to share information about working on the laptop or Bassi's environment.

One distinction between NERSC and Rice is that at Rice the problem was verbally presented in addition to the written presentation. At Rice we added a series of test cases that explored edge conditions to ensure the problem was solved. Passing all test cases and having the parallel code running on 8 processors faster than the serial version defined task completion. Some finished the task within a day and moved onto variations while others needed two days. Some subjects never completed the task. At the end of each day data was collected and backed up on a separate hard drive.

5. Analysis: The Boulder at the Top

All of the work thus far only pays off if you can analyze the data for the results you intended. Analysis happens for many reasons and on many levels. For illustration we summarize two analysis types: quantitative and qualitative.

Before beginning analysis we first unified and time aligned the sensor traces. For the Rice data we merged the Hackystat traces from the command line, shell and editors, along with web activity, Eclipse activity, and the per-minute human observation logs. The millisecond resolution, time ordered traces, were output to comma separated (csv) files for subsequent analysis.

Pre and post study survey data provided the information about the subject demographics and preparation as well as their perception of the study and the X10 language. These responses were summarized and used to provide context for the other analyses.

COMPLETION: Interleaving the time ordered data makes some analyses quite easy. Unambiguous completion criteria using the test scripts at Rice meant we were able to determine the time at which the problem was first solved successfully on each subject's laptop and on the 8-processor run on Bassi. This time, minus the sum of subject time away from the laptop, defined *time to completion* and served as the quantitative basis for comparing productivity between the two language conditions.

PATTERNS: Understanding programmer practice is important for understanding programmer use and perception of programming languages and tools. This requires a more qualitative approach. We began analyzing the event traces to determine the proportion of time spent in particular aspects of development (analysis, coding, debugging, etc.).

For the NERSC data we had only Hackystat and observation data. Analysis was done by hand by progressively segmenting the record into appropriate categories, based on what was in the data and our knowledge of programmer behavior and software engineering. For example a series of commands such as *edit, make, edit* iterated several times suggest cleaning up code (such as compiler errors). The sequence *edit, make, run, edit* suggests debugging. These sorts of manually derived categories formed the basis for our analysis of the Rice data.

VISUALIZATION: An important goal of examining the log of developer activities was to infer whether developers were following any type of consistent workflow. There are two reasons why such an investigation is interesting. First, many of the developers who write code for HPC machines are not trained software engineers or computer scientists, as was the case with some of our subject sample. Second, there are some additional activities required in the development of HPC code that are not present in the development of more traditional types of code, e.g. parallelization of code, debugging code running on multiple processors, and tuning the code/algorithm to increase performance on the parallel machine. Therefore, we expected to see some development patterns emerge from the visual analysis of the data.

In order to identify the workflow that a particular developer followed, we first needed to isolate the programming

events. Then we could analyze the order and frequency of these events. The programming events we were interested in were: *Writing New Code*, *Debugging Serial Code*, *Debugging Parallel Code*, and *Tuning Code Performance*.

To conduct this analysis we used an IBM proprietary tool, Zinsight, to view the various types of data described earlier in one screen. To identify which programming event occurred, we examined the commands typed at the command line or executed through the Eclipse interface. Specifically, we were interested in identifying when the subjects edited code, compiled code and executed code. When the subjects executed code, we were also able to capture the number of processors used (ranging from one to eight). While we could do this by reading the csv file in Excel, Zinsight made patterns easier to discern.

This analysis was conducted under the assumption that the sequence of commands would suggest the programming event that occurred. We began our analysis with the following hypothesized relationships:

- Edit → Make = *Writing New Code*
- Run (1 processor) → Edit → Make → Run (1 processor) = *Debugging Serial Code*
- Run (n processors) → Edit → Make → Run (n processors) = *Debugging Parallel Code*
- Run (n processors) → Make → Run (m processors) = *Tuning Code Performance*

As many qualitative hypotheses go, these hypotheses did not survive contact with the actual data fully intact. For this paper, we analyzed the development log of three subjects in detail. These three subjects exhibited different workflows, some of which coincided with our hypotheses and some did not.

Subject 1

We observed two patterns with subject one. First, most of his runs were done on two processors. It is not clear whether the lack of runs on one processor indicate that serial coding was not done in isolation, i.e. he went straight to parallel coding, or something else. The pattern we observed was that after a series of runs on two processors, he began systematically working up to more processors (i.e. three, four, five, etc…). Each time he added processors, he also edited the code. We assume the edits were to correct issues that became evident as more processors were added. At some point, this subject returned to two processors and started the process over again. We infer that the pattern this subject was following was: 1) Add new functionality; 2) Debug on two processors; 3) Add processors and debug; 4) Return to Step 1 on two processors.

The second pattern we observed in subject one was regarding his edits of source code. We were interested in the longer periods of editing, rather than quick fixes. These longer periods of editing fell into two groups: 1) the subject was running on more than two processors before and after the edit (the same number both times); 2) the subject was running on more than two processors before the edit, but only two after the edit. In addition, during many of the long editing sequences there are multiple 'make' commands executed before a 'run' command is executed.

Subject 2

The patterns we observed for Subject one were not as evident for Subject 2. We did not observe the same pattern of systematically running on more processors. Furthermore, for Subject 2, when he did a long editing session, it was not always followed by a run command. Sometimes it was followed by a make command and then more editing.

Subject 3

This subject again evidenced another pattern that did not exactly match those of Subjects one and two. Subject three performed most of his runs on two processors. There are only a few runs that used four processors and they are all at the end of the development cycle. It appears that this subject was not systematic about adding new functionality and then debugging that functional on multiple processors before moving on to new functionality. We also did not observe the same editing patterns for Subject three that we observed for Subject one.

This analysis is very preliminary and based only on three of the subjects who participated in the study. The next step of this work is to go further with this analysis to see whether these patterns are common across subjects or whether each subjects used their own style. The result of this analysis will allow us to identify HPC development patterns.

6. Discussion & Conclusion

The analyses we present here are only a few of those done; many more are possible. At Rice, only one of the MPI subjects finished the task, while five X10 subjects completed. Median time to completion for X10 subjects was a little more than half the time it took the MPI subject. In contrast, at NERSC all of the subjects using MPI finished, most of them in a day. Given those facts what do we really know from this study?

In part our knowledge is incomplete because the study itself is incomplete. We still need to run more experienced subjects using both languages. With each study we run we learn more about what it takes for a successful study. We cut the length of the study in order to appeal to more experienced (and time crunched) subjects. However, even with that and the backing of NERSC management we were still unable to recruit the number we needed.

Based on our earlier experience we added specific ending criteria and test cases in order to ensure that we knew when the problem had been successfully completed. We also changed how we observed programmer behavior to ensure coverage when programmers were not interacting with the computer. Both of these changes were successful,

providing us with more accurate data that we can now use confidently for more in depth analyses exploring programmer behavior on the way to completion.

There are things we still need to know. Students had trouble with MPI, even though they had parallel programming experience. This fits what we know from scientific computing where a programmer's first job is not computer science. While some of our subjects were engineers, some were also computer scientists, suggesting that other issues are involved. We need to figure out what those issues are.

In short, doing studies like this are difficult but rewarding. The more we do them, the closer we get to understanding what elements are important to help evaluate new programming languages and tools. Our hope is that the more we understand about the individual variability of programmers, as well as the similarities between approaches to parallel programming, then we will be able to evaluate and design tools to accommodate those issues.

Acknowledgments

We thank all our subjects and the others working with us on the PERCS project: Rachel Bellamy, Jonathan Brezin, Catalina Danis, Peter Malkin, John Richards, Cal Swart and John Thomas. Thanks also to Wim de Pauw who developed Zinsight and made it possible for us to look at our data visually. This work was supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.

References

[1] Bader, D. A., Madduri, K., Gilbert, J. R., Shah, V., Kepner, J., Meuse, T., and Krishnamurthy, A. http://www.ctwatch.org/quarterly/articles/2006/11/designing-scalable-synthetic-compact-applications-for-benchmarking-high-productivity-computing-systems/ Retrieved 8/31/09

[2] Basili, V. http://www.cs.umd.edu/~basili/papers.html. Retrieved 8/31/09

[3] J. C. Browne, T. Lee, and J. Werth, "Experimental Evaluation of a Reusability-Oriented Parallel Programming Environment," *IEEE Transactions on Software Engineering*, **16**(2), 1990, pp. 111-120.

[4] Danis, C. and Halverson, C. The Value Derived from the Observational Component in an Integrated Methodology for the Study of HPC Programmer Productivity. P-PHEC 2006, Austin TX. Pp 11-21.

[5] Ebcioglu, K, Sarkar, V., El-Ghazawi, T. and Urbanic, J. An Experiment in Measuring the Productivity of Three Parallel Programming Languages P-PHEC 2006, Austin TX. Pp 30-36

[6] HackyStat (www.hackystat.org/hackyDevSite/home.do). Retrieved 8/31/09

[7] Hochstein, L., Carver, J. Shull, F. Asgaril, S., Basili, V., Hollingsworth, J. and Zelkowitz, M. (Nov., 2005).A case study of novice parallel programmers. In *Proceedings of the 2005 ACM/IEEE SC/05 Conference*. IEEE, 2005.

[8] Hochstein, L ., Basili, V., Zelkowitz, M., Hollingsworth, J. and Carver.J. (September 2005) Combining self-reported and automatic data to improve effort measurement. ESEC/FSE 2005.

[9] Hofer, A. and Tichy, W.F. (2006) *Status of Empirical Research in Software Engineering.* http://wwwipd.ira.uka.de/~exp/otherwork/StatusEmpiricalResearch2006.pdf

[10] HPCS. High Productivity Computing Systems. http://www.highproductivity.org Retrieved 8/31/09

[11] Istanbul. http://live.gnome.org/Istanbul Retrieved 8/31/09

[12] Lawrance, J. (2009) Unpublished Dissertation and personal communication.

[13] Nystrom, N.A., Urbanic, J., and Savinell, C. Understanding Productivity Through Non-intrusive Instrumentation and Statistical Learning. P-PHEC 2005, San Francisco.

[14] Nystrom, N., Weisser, D. and Urbanic, J. The SUMS Methodology for Understanding Productivity: Validation Through a Case Study Applying X10, UPC, and MPI to SSCA#. P-PHEC 2006, Austin TX. Pp 37-45

[15] PERCS. Productive Easy-to-use Computing Systems. http://www.research.ibm.com/hptools Retrieved 8/31/09

[16] Perry, D.E., Staudenmayer, N.A., Votta, L.G. (1994) *People, Organizations and Process Improvement*. In IEEE Software, July. Pp 36-45

[17] Perry, D.E., Staudenmayer, N.A., Votta, L.G. (1995) *Understanding and Improving Time Usage in Software Development*. In Process Centered Environments. Fuggetta and Wolf, Eds. John Wiley and Sons Ltd.

[18] Schneiderman, B. and Carroll, J. (1988) Ecological studies of professional programmers. Communications of the ACM. 31(11) ACM.

[19] Shull, F, Carver, J., Hochstein, L. Basili, V. (2005) *Empirical study design in the area of High-Performance Computing.* 4th International Symposium on Empirical Software Engineering (ISESE '05). November 2005.

[20] Slogger : https://addons.mozilla.org/en-US/firefox/addon/143 Retrieved 8/31/09.

[21] X10 programming language. www.research.ibm.com/x10/; http://en.wikipedia.org/wiki/X10_(programming_language); http://x10.codehaus.org