Towards an Ecologically Valid Study of Programmer Behavior for Scientific Computing

Christine A. Halverson, Cal Swart, Jonathan Brezin, John Richards, Catalina Danis,
IBM, TJ Watson Research Center
PO Box 704 Yorktown Heights, New York 10598 USA
1 914 784 7300
{krys, cals, brezin, ajtr, danis} @us.ibm.com

ABSTRACT
This paper presents the motivation, design rationale and implementation detail of a study of programmer behavior in scientific computing circa 2002. We discuss the constraints of creating a retrospective baseline—the methods used, the necessary conditions—and what we have learned. We examine the problems of doing such a study and the difficulties of ecological validity in a partly controlled setting.

Categories and Subject Descriptors
D.2.8 [**Software Engineering**]: Metrics – *performance measures*

General Terms
Empirical Studies, Measurement, Design, Human Factors.

Keywords
Empirical studies, metrics, measurement, productivity.

1. INTRODUCTION
Scientific computing, and its use of high performance computing systems (HPCS), covers a wide area. Variations include the range of problems being solved within a variety of organizational structures: academic, governmental and corporate. Within these organization types there is more variation in how HPC is supported. Program teams may be as small as one and as large as 30 or more. Code construction itself ranges from single language to multi-language (including scripting languages) and in size from so-called *Kleenex codes* to large multi-module codes. Scientific computing has a lot of issues in common with *traditional* software (i.e. sequential code). As we shall see, it also has many that are specific to it.

The motivation for this work comes from our participation in the DARPA sponsored High Productivity Computing Systems (HPCS) program. We are currently in Phase 3 of an eight-year effort aimed at developing peta-scale machines and supporting tools that significantly improve the productivity of the scientists, programmers, data managers and system administrators who will use them.

Our focus here is on programmer productivity. Decades of work in software engineering have led to a number of theories and findings. These do not necessarily apply however, to scientific computing. As Shull et al [24] point out many of the underlying assumptions pertinent to traditional software development do not hold for scientific computing. Just one example is the importance of machine specifics in the programming and running of codes. Thus there are issues unique to understanding productive programmer behavior writing sequential code, and there is a whole new set of problems presented by the need for parallel coding.

How to define productivity is itself not so simple. Is productivity measured in relation to an individual, a team or a specific project? What kinds of measures go into evaluating productivity? Is productivity related to work done by the programmer? Or is it the overall monetary gain from a particular project? While we are aware of these difficulties they are not our immediate concern. We are currently working with a modified economic model (utility equals work divided by cost). What is more important for us isn't what measurements are chosen and how they are calibrated. Evaluating a change in productivity—for example increased productivity because of tool use—requires having an established baseline that can be compared against.

For this project the baseline is defined to be 2002. Unfortunately no one measured or recorded parallel programming behavior at that time. To establish that baseline we need to capture programmer behavior using operating environments and tools that are representative of what various communities doing this sort of computing actually used at the time. For this reason we are working towards an ecologically valid study design. Ecological validity refers to how close the method, setting, and materials mimic the real world situation, that is real world in 2002. In addition, the data we collect must also be appropriate for comparing to similar tasks using new tools as they will be performed circa 2010.

We shall map out here how a range of measurements and techniques can be brought together in an integrated methodology See Danis and Halverson [8] for a more detailed picture of programmer behavior. In what follows we present the design of an empirical study focused on baseline behavior in 2002. Our design is working towards ecological validity within the scope of what is possible and our deadlines. We discuss several data collection techniques to collect measures that we will use to provide a baseline of use against which new tool use can be evaluated. We begin with a brief summary of related work. Section three focuses on our empirical design and discusses the issues and trade-offs leading to particular decisions. Section four presents a brief scenario of the study in action and issues are raised for discussion in section five.

2. RELATED WORK

Previous efforts to understand programmer behavior in general, including productivity, 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. [21, 22] for more field based approaches; Hofer and Tichy [16] for a review of empirical approaches in the last decade; and Basili et al [4] for lab based empirical approaches.)

Previous efforts to understand parallel programming behavior have used the same methodologies, but those efforts have been comparatively limited. [10]). A recent trend has been so called *hybrid* or integrative methods; that is studies that use several methodologies in order to triangulate on a more accurate understanding of behavior.

2.1 Studies in Software Engineering

There is a rich history of empirical studies of programmers and programming since the 1960s. These studies fall into a number of disciplines including psychology, computer science education, human factors, information technology, and software engineering (SE). Not surprisingly the domain of the researchers tends to focus the study on those aspects central to their field as it occurs during sequential programming practice.

Software engineering has gradually adopted an empirical model of validation of new software concepts and tools, largely drawn from psychology experiment models. Zelkowitz and Wallace [30] show the increase in empirical validations in the literature from 1985 1995. Sjoeberg et al [27] go further, focusing on the rise of controlled experiments in the literature from 1993-2003.

Productivity in computing was often equated to work, a natural outcome of the recognition that application development was taking too long and requiring greater human resource costs than hardware costs. Thus metrics such as SLOC[1] have been used as a measure of programmer work and therefore productivity. (More code does not necessarily equal more productive programming. See [2] for some problems with using LOC as a metric). Various measures related to completed lines of code have been suggested and discredited. Subjects ranged from students to professionals and studies used a wide range of methods from somewhat controlled experiments to in depth field observations of programmers.

2.2 Studies in HPCS

There have been studies in HPCS since the early 80's. Some early studies centered on specific machines (e.g., LeBlanc et al [18]), while others focused on programming languages or parallel environments (see Browne et al [5]). With the DARPA HPCS program there have been a number of proposals redefining productivity in terms of an economic model (e.g. see Snir and Bader [26]. However, underlying all of these is programmer behavior and how to map that to metrics that can be used for evaluating productivity.

The manner of data collection and the available subject pool have largely constrained the method and measurements. As most studies occurred in universities the available subjects were students, usually in their first parallel programming class (e.g. Hochstein et al, [14]). The programming task was generally one assigned to the class, such as Sharks and Fishes [15], and the class also dictated the programming language. Editors for coding and machines for running were determined respectively by personal preference and availability. Hochstein et al offers an analysis of data gathered from 4 graduate classes at different institutions programming classroom exercises. They point out the problems of trying to compare across different languages and machines, something that is not an issue in traditional software engineering.

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 [15] combined automated data gathering using Hackystat [11] with manual data provided by programmer self report as a *hybrid* method.

We have used a different combination for a hybrid – the integrated methodology [8]. This method combines automatic data collection (SUMS, [19, 20]) 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 interruption-causing context switching by the programmer and the potential for conflict of interest in the content of the programmer's self reports.

2.3 Ecological Validity

Methodological differences aside, one of the pervasive problems in empirical studies of programmers is how ecologically valid the study is. Ecological validity refers to how close the method, setting, and materials mimic the real world situation. Controlled experiments are by nature set up to limit the variance in what is being studied. This often means the circumstances of the study are so artificially controlled that they cannot be ecologically valid.

The recognition of the necessity for ecological validity in programmer studies is not new. In a special section of CACM in 1988 Schneiderman and Carroll [23] focused on the need for ecological studies of professional programmers in their native environments. The series of studies done by Perry et al picked up this call again in the mid 1990s [21, 22]. In both cases researchers were figuring out ways to study programmers in the wild resulting in qualitative and quantitative data. However studies like these are not always practicable.

To illustrate the difficulties here are just a few examples of the variability and other issues that are related to our study. Individuals customize their environments, making evaluations between individuals difficult. In some cases our subjects may work on classified work at least part of the time and the organizational culture would have problems with us adding sensors that record programmer actions on their machine. Thus our goal for ecological validity must walk the line between what is realistic for the setting and what is realizable for the study.

An additional issue related to this is the amount of data to capture and how to capture it. Unfortunately the ideal—collecting all pertinent data of programmer behavior and context in the real world over the life of a project—is extremely difficult as well as

[1] Source lines of code

economically impossible. Our goal then is to develop a method that provides us with the best data to understand productivity as it pertains in the world of scientific computing in a realistic and cost efficient way. In the following section we present our approach using the integrated method and striving for ecological validity.

3. OUR APPROACH

The context of software development in scientific computing is not homogeneous. From previous research we know that scientific computing varies on several dimensions: organizational context, characterization of code developed, programmer resources, and code life.

In our field observations and interviews, coupled with the literature, we find the following categories out of probably many more.

1. Organizational context: Academic, Governmental, and Corporate.
2. Code Type: New code, new feature in old code, code maintenance, code refactoring, code reuse.
3. Problem characteristics: e.g. high data throughput, computationally intense
4. Programmer resources: individual or team dedicated to code over a significant part of the code lifetime
5. Code life: days, weeks, months, years, decades.

The DARPA HPCS Productivity effort addressed some of this variability by developing a number of workflows that captured scenarios of development. The most straightforward workflow to address is that of a solo programmer writing code from scratch. Perhaps the most difficult is a large team developing, and maintaining a large multi-module code that has lasted over decades. Steps to ensure ecological validity must vary with the circumstances surrounding each of these scenarios.

We opted to start focusing on an individual, programming a realistic problem that would be tractable in a reasonable amount of time (no more than 2 days) and using the environment and tools available in 2002. In the sections that follow we delve into each of these areas.

Building on our earlier work [7,8] we are again adopting an integrated methodology, but with some adjustments. In that study we learned that given a limited amount of resources (trained observers), it is better to observe a small number of subjects completely rather than sample across all subjects. Thus, since we cannot observe every subject in detail we have placed more emphasis on non-intrusive automated data collection than direct observation. However, a subset of subjects will also be observed and video taped as a point of validation for the method.

3.1 Study Design & Tradeoffs

Unlike many studies our focus is not on measuring time to solution. Rather it is to make sure that we see the usage detail necessary to understand how a particular tool affects programmer behavior and thus his or her productivity. Except in the case of language comparisons, how fast the code runs is mostly a factor of the machine and is not in this case pertinent.

3.1.1 Overall design

We have taken a ‘soup to nuts’ approach, covering a full development effort from conception of the problem solution, to coding, testing with provided data, and if time, tuning for better performance. The setup for studying a single programmer seems straightforward: one individual, one problem, one personal computer (PC) and one supercomputer (SC). This simplicity however obscures the many decisions and trade offs along the way.

3.1.2 Problem

The ideal problem would be one that needs to be solved in real life. However, there are a number of issues with using that kind of problem. For example, some are very domain specific, requiring deep domain knowledge to even understand the problem. In some cases the solutions may be proprietary, or need to be protected in some way, so it is difficult to get permission to collect detailed data during a regular work process. Nonetheless, the problem must be sufficiently realistic for comparison to actual practice. In addition, what we are most interested in are those aspects of programming that may be helped or hindered with tools. Thus the difficult aspect of solving the problem, something that could take weeks or more, is something that we want to minimize. Finally, time limitations require a problem that is tractable but not trivial.

Our solution was to use one of the problems developed for the HPCS program called a Synthetic Scalable Compact Application (SSCA). (For details see Bader et al. [3]). SSCA1 presents a problem of pattern matching, the Smith-Waterman algorithm using for example gene sequences. While some domain background in the problem description grounds it in genetics, the problem does not require deep domain knowledge to solve. We provide them 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 Unlike SSCA2, SSCA1 can be solved cleanly using MPI, the dominant means in 2002, in exactly the way it was designed to be used.

3.1.3 Machine Issues

Our objective in this study is to establish a reasonable baseline of programmer behavior circa 2002 by studying programmers doing a common parallel programming exercise using the kind of machine and programmer tools then available**.** This means we need to use the appropriate operating environments, tools and languages.

We were fortunate to begin our study design on a machine that was close to 2002 capabilities. The National Energy Research Scientific Computing (NERSC) center’s IBM SP RS/6000 Power3—Seaborg—was brought online in 2001 and still had the software and tools that fit our needs. As we cannot turn the clock back to 2002 we believe that setting up a study of programmer behavior under these conditions is the most reliable way to establish a realistic baseline for future comparisons.

Our pilot subject used Seaborg until it was decommissioned in January 2008. We switched our setup to Bassi—an IBM p575 Power5 system. While the computational capabilities—from chip design to machine architecture—are somewhat different, in all the ways that mattered for this study they were the same. That is to say that we were able to provide the same software stack (operating system, editors, mpi library and compile commands) as we had on Seaborg. Our study subjects all used Bassi, removing the problem of comparing across machines.

3.1.4 Laptop Issues

Previous field research and interviews indicated that many programmers program directly on the interactive portion of a machine's nodes. However we have seen those who develop code on a local machine and then upload and run the code on a supercomputer [12]. For the purposes of the study laptops are just as powerful as a desktop and much easier to move around. In addition, by dedicated laptops we are able to install a base configuration that can subsequently be used by the new programmer tools necessary for the 2010 comparison.

We set up five identical ThinkPad T61p laptops for this study. The base configuration required on the laptop was constrained by three factors:

- the requirements of the automated data collection software (Hackystat);
- what tools were available in 2002 and;
- the base requirements of the tools being developed.

Table 1 shows a summary of the operating environment and tools used to configure both the laptops and Bassi for the study.

Table 1.

	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. Limited to NERSC and MPI sources	none
Automated Data Collection	Hackystat Slogger Istanbul	Hackystat

3.1.5 Subjects

The range of contexts where parallel programming happens argues for a range of subjects. We know that some organizations have dedicated interdisciplinary teams with experienced programmers, while others are constantly bringing in college graduates with little parallel experience and seeing a large attrition rate [12]. To address this we chose to focus on two levels of experience. Experienced subjects are defined as having 10 or more years of experience in parallel programming, while novices were considered to have had at least one parallel programming class and 3 years of experience programming. In this way we hope to avoid the problem of separating out effects caused by just learning parallel programming.

Finding subjects with these levels of experience, who have the free time for such a study, is not trivial. We had no expectation that we could recruit enough subjects in order to demonstrate statistical validity and generalizability. Instead our focus is a detailed analysis of a relatively small number of subjects resulting in quantitative and qualitative results.

We proposed to recruit 10 subjects in each of the experience conditions. Allowing for dropouts we hope to have a minimum of 8 in each. Why so small a number? In addition to the overall programming experience, all subjects needed to be well versed in the tools and languages circa 2002. There are other questions we would like to explore; such as whether being trained in a science rather than computer science leads to different programming behaviors or productivity measures. However, having already narrowed the pool of possible subjects by imposing the requirements for ecological validity we felt that any additional requirements would make the pool too small to recruit from successfully.

3.1.6 Data Collection and Recording

In previous efforts we worked with the Pittsburgh Supercomputing Center in a comparative language evaluation [8,9]. In that study we used an integrative methodology gathering data in three ways: automatically collected computer interaction data via the PSC SUMS application [19, 20], detailed behavioral data noted by trained observers, and, in some cases, video taped data. SUMS collected extensive data, including recording the command line interface (CLI) and application feedback displayed on the screen, web browser activity, and a snapshot of the code every 10 minutes. This provided us with almost more data than we could use. PSC used machine learning techniques on the automatically collected data in order evaluate it. Our observations were used to fill in gaps where no automated data was collected and to infer programmer intent in some cases.

We have since moved to using the Hackystat7 framework for automated data collection. Hackystat [11] 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.

Hackystat is designed to be extensible and configurable along three primary dimensions: (1) the set of "sensors" (i.e. plug-ins to development tools that gather process and product data); (2) the set of "sensor data types" (i.e. structures that represent raw sensor data of various types); and (3) the set of "applications" (i.e. server side analyses that provide useful summaries of developer behavior over time).

To provide this flexibility, Hackystat has an architecture consisting of over 60 public *modules* that are organized into four *subsystems*. The *Core* subsystem includes modules that provide basic framework mechanisms. Modules in the *Sdt* subsystem implement sensor data types. Modules in the *Sensor* subsystem implement sensors for development tools. Finally, modules in the *App* subsystem provide applications that generate useful analyses over the sensor data collected by the sensors. As it was originally designed for software developer to collect data on themselves the sensors tend to focus on current tools and systems. This makes using Hackystat attractive for instrument our 2010 tools, but unfortunately means that we cannot take advantage of the full wealth of sensors.

In our case, we attach the sensors to the development tools in each laptop in advance and configure the laptop to automatically push the data out to the server at the end of the programming session. Like SUMS we can collect data from both the command line and the editors. Unlike SUMS we do not capture the compiler responses, nor do we have code snapshots over time or web browser history. We use another piece of software—Slogger [25]—to record web browser activity. This data is integrated with the Hackystat data after the study is complete.

As we noted above, detailed observations as we did in our previous study are extremely time-consuming. We wanted to move to something that would be less person intensive. Video data, while easy to record, requires fairly obtrusive equipment. Our solution is to move to screen capture software to supplement the Hackystat and Slogger data. While this data, like video, requires time to analyze it does have the benefit of being unobtrusive and automatic.

We use an open source project called Istanbul [16] for screen capture. This does require a few additional steps to make sure it is recording on each laptop, so it is not as unobtrusive or automatic as Hackystat. It produces a file that uses the open source video codec Ogg Theora. It can be viewed with both its own application and VideoLAN [29] an open source player that handles Ogg Theora.

3.1.7 Validation

Validation of this method is necessary along two dimensions. First the study design and its details must be validated through a pilot study. Separate from this is the validation of the various data collection methods used. That is, are we getting the coverage of data we need and is it sufficiently detailed and complete? This was particularly important, as this was our first use of Hackystat.

We designed our pilot study to collect multiple sources of data that could then be compared against each other. In this way we are able to verify the accuracy and sensitivity of the data collected by Hackystat as well as where there are gaps that need to be addressed by another method.

3.2 Execution

3.2.1 Recruiting and qualifying subjects

We are lucky that our association with NERSC, and the proximity of Lawrence Berkeley National Laboratory (LBNL) and University of California, Berkeley (UCB) provide an appropriate pool of subjects. The scientific computing world is compact and overlapping within these three organizations. Our strategy was to have a recruitment email sent out by the Associate Director of NERSC targeting NERSC, the Computer Research Dept. (CRD) at LBNL and the various outreach efforts into the larger community. In addition we targeted several of the professors of graduate level parallel classes at UCB.

Once subjects express an interest based on the rough qualifications outlined in the recruitment email we send them an additional survey to gather information about their experience working in parallel programming, including classes, projects for fun and paid work. If subjects qualified they were then scheduled for a two-day session at an IBM office in San Francisco, CA.

3.2.2 Machine setup

The computational environment necessary for scientific computing varies in both hardware and software. Having determined the closest hardware (Seaborg) we had to do two things. First we needed to evaluate what was currently on Seaborg and how it compared to what was available in 2002. In most cases this was a question of the software level available. While we could not take software back to an earlier level we were able to assure ourselves that the existing versions varied by an acceptable amount. In addition we investigated which software predominated in use. For example, TotalView [28] was the most commonly used debugger, with some use of both dbx and gdb reported in the annual user surveys. However, the exact usage was not tracked. In contrast library use was tracked.

In addition, some software updates needed to be made to Seaborg in order to use Hackystat to directly collect data from interactions on the machine. These included the version of a Java compiler necessary for Hackystat to run, as well as ensuring the levels of Vim and bash were the ones supported by Hackystat. (When we transitioned to Bassi we needed to duplicate these efforts.)

On both the laptop and Bassi we provide a directory that includes a serial coding of the problem and a range of data sets it can be run against. That code is capable of being compiled and run on both machines. We duplicate this in another directory where we ask them to work on their mpi code. We also provide a make file and documentation about the problem. Each subject has a dedicated account on Bassi that corresponds to their laptop name for easy identification.

3.2.3 Task issues

The task or problem needed to be stated in a way that would be sufficiently clear without giving away the answer. We had used the same problem in our previous study, however in that case a domain expert at PSC presented the problem. He was also available to answer questions for an additional half hour. In addition the study schedule provided a considerable amount of time for subjects to think about the problem, including an hour or so after the afternoon presentation, and overnight before beginning coding the next morning.

In contrast we needed to present the problem in such a way as to reduce the time necessary to understand the problem and its potential solution. Needing to maximize the time available for coding the problem motivated us to make it as straightforward as possible.

3.2.4 Pilot

We did a pilot study in order to work out the details of the study setup and to verify that we were collecting the appropriate data. Our pilot subject was a NERSC retiree with many years of experience in the HPC world. During the pilot we worked out details of the interaction between machine configurations. We were also able to refine the problem presentation significantly based on his feedback. As he coded and parallelized the Smith-Waterman algorithm we recorded data with Hackystat and Slogger, observable programmer behaviors were noted by a trained observer, and recorded the complete session on videotape. Data analysis is now underway to detail where our measurements are sufficient and where we need to look for additional data.

4. Study Scenario: An Example

Once subjects are screened and scheduled, then the study can begin. In this section we briefly outline the various stages and progress of the study.

4.1 Main study

For each session a setup process occurs to make sure all the instrumentation is properly set up to record data. We can accommodate up to 5 subjects at one time, although in practice it is usually less. Once the subjects arrive and get settled in we cover informed consent before beginning the main study. First we make sure that everyone understands that the study is about patterns of behavior rather than their skill at solving the problem. We introduce them to the setup on both the laptop and Bassi—such as where files are located—and then let them begin reading the problem. Should there be any questions regarding the problem statement itself we have the author available to answer questions.

The subjects begin working on the problem whenever they feel ready. We provide them with a serial version of the code written in C (on the laptop and as a printout). They have pad and paper as well as a cheat sheet that summarizes the login commands for Bassi and the run commands for the serial version of the code. Almost all of their computer interactions are captured: command line interface (CLI), editor, and web use. What is not captured is their time not involved with the computer. In this case, where we are largely looking at patterns of behavior interacting with tools and in various coding phases, the lack of the other time is acceptable. All subjects also have their screen recorded. For additional validation one subject in each session is also video recorded.

With multiple subjects in the same session we also get audible interactions that are captured on videotape. Subjects are cautioned not to confer about the problem but are able to share information about working on the laptop or Bassi environment.

Most subjects spend an hour or so reading the problem, and reviewing the existing code. Some may take notes, or draw diagrams related to the solution. As some programmers think better on their feet we allow them to get up and walk around the building to think. An observer records the times they leave and enter the session room. (This is possible because we have so far run the study on weekends and the floor has been empty and isolated.)

Everyone takes a break for lunch (about 45 minutes) and then returns to the task. Some finish within a day while others return for a second day. At the end, after collecting some additional information and taking a post session survey the subjects may leave. Rather than have them fill out the post session survey there most subjects opt to have it emailed. They fill out the document and email it back.

5. LESSONS LEARNED

So far we have recruited four subjects, three of which have completed the study. (The fourth is being scheduled). It is too early to present results, as data analysis has just started and is ongoing. However there are a number of things we have learned from our experience.

5.1 It Always Takes Longer

Most academics, and even corporate researchers, are familiar with gaining the approval of their home institutional review board (IRB) for a study using humans. In this case however, several things complicated the process. As corporate researchers we do not have the same requirements for CITI [6] certification required by many academic institution. Getting our team certified added extra time to the process. In addition we needed to pass two levels of review: LBNL and their parent UCB. While this was not difficult it was extremely time consuming taking about six months.

Switching machines added another big chunk of time. Even though Bassi was mostly up to date in the versions of compilers and applications, we still needed to verify each one.

5.2 Recruitment and Experience

Subject recruiting was harder than we expected. In our previous study we had found that DARPA HPCS participants were unwilling to dedicate their personnel's time for a full week. In this case there were two differences. First, we had reduced the scope of the study to two days for coding the problem. Second, we had buy-in from NERSC management who agreed to let individuals choose to do the study on work time, or they could do it on their own time and get paid. For graduate students we thought that offering to pay for two days, even if they finished in one, would be added enticement. Neither of these worked exactly as expected.

- We required subjects to travel to a nearby location that was easily accessible to public transportation. However, some expressed that this was too much to bother with.
- When we received IRB approval it was the holiday season (between Thanksgiving and Christmas) before we were able to recruit subjects. Not surprisingly many potential subjects had plans.
- Students had some difficulty scheduling two days in a row due to classes or meetings.

One issue we didn't expect was related to experience. We did not realize that experienced programmers might not be programming regularly as part of their job. In particular several subjects were in management and expressed how their skills were rusty, requiring them to take time looking up MPI calls and other details. On the other side, many of the comparative novices had lots of experience, but not necessarily with MPI—further reducing the available subject pool.

Finally, also related to experience, we were surprised that the "obvious" solution to the subjects was the harder wave front approach. In retrospect this makes sense, as the scientific programmer's job is to make sure that their code handles all the edge conditions that could occur in the phenomena they are modeling. It was an awkward situation to remind subjects that, as one participant put it: "Things don't have to be perfect. It's ok if the monkey dies."

5.3 Old Habits are Hard to Break

Vi was the first visual editor for UNIX. It debuted in 1976 and is still in use, although almost no one uses a true vi anymore, except perhaps in scientific computing. This is not because they are all using emacs. It's because a number of offshoots have been created that will work on operating systems other than UNIX. The most popular of these is Vim, covering many laptop operating systems as well as VMS. Hackystat's sensors only work with Vim, not with vi. Telling subjects to use Vim, and even reminding them, is not enough to overcome ingrained habits. Our solution is simple: alias vi to call Vim.

5.4 A Matter of Control?

As we mentioned earlier, we provided a working make file to compile the program. Generally make files are perceived to help the programmer by capturing all the routinely repeated, and often arcane, actions necessary to build a program. We justified this decision based on our experience that each supercomputer center has very detailed and explicit profiles that need to go into make files, leading at a minimum to providing templates.

Counter to our expectations almost no one used the make file. The majority ran things manually from the command line. We hypothesize that this may be due to their perception that this is (comparatively) a toy problem. In that case we must ask ourselves how we might capture the various issues that may only be exposed in a more complicated example.

5.5 Accounts and Bassi, and Laptops! Oh My!

Resolving differences between machines raised some surprisingly frustrating problems. We were lucky that all of our subjects had already used Bassi, so they were familiar with it. In our pilot we had trouble finding the right terminal type to interact with Seaborg from the laptops. Several we tried produced extra characters on one machine or the other. Luckily we discovered the right terminal type and subsequently how to set the terminal type in order to get syntax highlighting in Vim. All of this translated to Bassi. However, Vim on the laptop continued to produce artifacts in the code that sporadically appeared after uploading to Bassi. We may never figure this out, but the participants still need to manually correct them.

Issues that experienced programmers in this world take for granted are things that we as researchers might not consider. Take for example login accounts. With our pilot subject, the retiree, they reactivated his prior account and had him change his password. When he went to log in however the password did not work. It seems the password change had not propagated around all the nodes. He got around this by logging in on an administrative node, not something we could do with subjects. When we moved to Bassi we were careful to check each of the accounts for this problem. Here however, it was not the password but the permissions. A user could do anything – except submit a job.

That was fixed and we had our first session. One of the subjects was involved with managing user interactions with the machine and was having trouble submitting a job. We suspected the previous problem with permission, but it was not the same error. Instead it was an oversight—we had forgotten to ask for a portion of the queue on Bassi to be set aside for our use. This in turn led to another problem. One subject who wanted to do it all on Bassi, using the laptop as a terminal, found he could not because the queue was full. Other subjects, who were already doing everything on the laptop, helped him with the appropriate compile commands and configuration to run the parallel version on the laptop. In the end we were able to resolve the queue issues and two subjects completed and ran their parallel versions on Bassi.

6. DISCUSSION

What we have presented so far are the details and issues with setting up and executing an empirical study of programmer behavior in scientific computing practices. On the surface we report on the common problems of studies: things will always take longer and expect the unexpected. As many in this community do not have direct experience setting up and executing these studies we consider it valuable to pass on the details of our experiences. There two meta level issues that merit further consideration.

6.1 A Retrospective Baseline?

What does it mean to establish a baseline that is retrospective? Without a time machine we can at best partially duplicate the circumstances and environments of a particular time. We were lucky in that much of scientific computing still lives on the command line with basic editors, making it easy to come close to an effective environment. In an environment like NERSC a lot of effort has been put into making sure that programming environments are as consistent as possible across machines and time, so we are reasonably assured that any impact from the parallel environment has been minimized.

Nonetheless, as Shull et al [24] point out, hardware variations matter in HPCS software development, as do the types of developers, their process and the available resources. We have tried to limit resources that seem essential today (such as Google) and worked to minimize differences where we can. While we strive for ecological validity there is only so far we can go.

6.2 Ecological vs. External validity

At the beginning of this paper we argued for pursuing ecological validity rather than external validity. In part our motivation is recognition that previous studies that focused on beginning parallel students are not really indicative of the problems in actual practice. The question is: are we sacrificing external validity for ecological validity?

One answer is to some extent yes. Ecological validity puts greater constraints on study design, setup and subject pools. These constraints further reduce the overall numbers of available subjects necessary to establish external validity. The contrast is student subjects are more plentiful, but their behaviors and expectations are different. Striving for the most realistic programming situation is worth it because of the greater verisimilitude of observed programmer practice to actual practice in scientific computing.

7. CONCLUSION

8. ACKNOWLEDGEMENTS

Deep thanks to the others working with us on the PERCS project: especially John Thomas, Rachel Bellamy, and Peter Malkin.
This work was supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002.

9. REFERENCES

[1] Albrecht, A. J. (1979) "Measuring application development productivity," in *Proc. Joint SHARE/GUIDE/ IBM Appl. Development Symp.* Oct. 1979, pp. 83-92

[2] Armour, P. (2004) Beware of Counting LOC. *Communications of the ACM*. 47(3) pp 21-24

[3] 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 February 21, 2008.

[4] Basili, V. http://www.cs.umd.edu/~basili/papers.html. Retrieved Feb 27, 2008.

[5] 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.

[6] CITI. Collaborative Institutional Training Initiative. http://www.citiprogram.org/. Retrieved February 28,2008.

[7] Danis, C. (Nov, 2006). *Forms of collaboration in high performance computing: Exploring implications for learning.*

Proceedings of the conference on Computer Supported Cooperative Work, Banff, CA.

[8] Danis, C. and Halverson, C. (Feb, 2006). The Value Derived from the Observational Component in an Integrated Methodology for the Study of HPC Programmer Productivity. In *Proceedings of the Third Workshop on Productivity and Performance in High-End Computing.* Held in conjunction with the Twelfth International Symposium on High Performance Computer Architecture, Austin TX. Pp 11-21.

[9] Ebcioglu, K, Sarkar, V., El-Ghazawi, T. and Urbanic, J. (2006) An Experiment in Measuring the Productivity of Three Parallel Programming. Languages. In *Proceedings of the Third Workshop on Productivity and Performance in High-End Computing.* Held in conjunction with the Twelfth International Symposium on High Performance Computer Architecture, Austin TX. Pp 30-36

[10] Eccles, R. and Stacey, D.A. (May, 2006) Understanding the Parallel Programmer. In *Proceedings of the 20th International Symposium on High-Performance Computing in an Advanced Collaborative Environment (HPCS'06).* IEEE: pp 2-12

[11] Hackystat (www.hackystat.org/hackyDevSite/home.do).

[12] Halverson, C. Unpublished Fieldnotes.

[13] Hochstein et al 2003

[14] Hochstein, L., Carver, J. Shull, F. Asgari, 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.

[15] Hochstein, L., Basili, V., Zelkowitz, M., Hollingsworth, J. and Carver. J. (September 2005) Combining self-reported and automatic data to improve effort measurement. In *Proceedings of the Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering* (ESEC/FSE 2005).

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

[17] Istanbul. http://live.gnome.org/Istanbul Retrieved February 28, 2008.

[18] LeBlanc, T., Scott, M., and Brown, C. (1988). Large-scale parallel programming: experience with BBN butterfly parallel processor. *ACM SIGPlan Notices.* 23(9) pp 161-172.

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

[20] Nystrom, N., Weisser, D. and Urbanic, J. (Feb, 2006) The SUMS Methodology for Understanding Productivity: Validation Through a Case Study Applying X10, UPC, and MPI to SSCA#. In *Proceedings of the Third Workshop on Productivity and Performance in High-End Computing*. Held in conjunction with the Twelfth International Symposium on High Performance Computer Architecture, Austin TX. Pp 37-45

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

[22] 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.

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

[24] 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.

[25] Slogger: https://addons.mozilla.org/en-US/firefox/addon/143 Retrieved February 24, 2008.

[26] Sjoeberg, D., Hannay, J., Hansen, O., Kampenes, V., Karahasanovic, N., Liborg, K, and Rekdal, C. (2005). A Survey of Controlled Experiments in Software Engineering. IEEE Transactions on Software Engineering 31(9): 733-753.

[27] Snir, M. and Bader, D. (2004) *A Framework for Measuring Supercomputer Productivity.* The Intl. Journal of High Performance Computing Applications. 18 (4) pp 417-432

[28] TotalView. http://www.totalviewtech.com/index.htm. Retrieved February 28, 2008

[29] VideoLAN. http://www.videolan.org/. Retrieved February 28, 2008.

[30] Zelkowitz, M, and Wallace, D. (1998) Experimental Models for Validating Computer Technology. IEEE Computer 31(5), (May, 1998) 23-31