Proceedings of the Third Workshop on Productivity and Performance in High-End Computing (PPHEC-06)

February 12, 2006
Austin, USA

Held in conjunction with the
Twelfth International Symposium on
High Performance Computer Architecture

Program Chair:

Ram Rajamony, IBM Research

Program Committee:

Philip Johnson, University of Hawaii
Nick Nystrom, Pittsburgh Supercomputing Center
Ram Rajamony, IBM Research
Vijay Saraswat, IBM Research
Lawrence Votta, Sun Microsystems

3^{rd} Workshop on Productivity and Performance in High-End Computing
Febryary 12, 2006
Austin, USA

Yes, There Is an "Expertise Gap" In HPC Applications Development 5
Susan Squires, Michael L. Van De Vanter, and Lawrence G. Votta

The Value Derived from the Observational Component in an Integrated Methodology 11
for the Study of HPC Programmer Productivity
Catalina Danis and Christine Halverson

Parallelization of a Molecular Modeling Application: Programmability 22
Comparison Between OpenMP and MPI
Russell Brown and Ilya Sharapov

Implementing the CG and MG NAS parallel benchmarks in X10 29
Vijay Saraswat

An Experiment in Measuring the Productivity of Three Parallel Programming 30
Languages
Kemal Ebcioğlu, Vivek Sarkar, Tarek El-Ghazawi, and John Urbanic

The SUMS Methodology for Understanding Productivity: Validation 37
Through a Case Study Applying X10, UPC, and MPI to SSCA#1
Nick Nystrom, Deborah Weisser, and John Urbanic

Proceedings of the Third Workshop on Productivity and Performance in High-End Computing (PPHEC-06)

Yes, There Is an "Expertise Gap" In HPC Applications Development

Susan Squires
Sun Microsystems, Inc.
15 Network Circle UMPMK15-204
Menlo Park, CA 94025
1-650-786-3441
susan.squires@sun.com

Michael L. Van De Vanter
Sun Microsystems, Inc.
16 Network Circle UMPMK16-304
Menlo Park, CA 94025
1-650-786-8864
michael.vandevanter@sun.com

Lawrence G. Votta
Sun Microsystems, Inc.
16 Network Circle UMPMK18-216
Menlo Park, CA 94025
1-650-786-7514
lawrence.votta@sun.com

ABSTRACT

The High Productivity Computing Systems (HPCS) program seeks a tenfold productivity increase in High Performance Computing (HPC), where productivity is understood to be a composite of system performance, system robustness, programmability, portability, and administrative concerns. Of these, programmability is the least well understood and perceived to be the most problematic. It has been suggested that an "expertise gap" is at the heart of the problem in HPC application development. Preliminary results from research conducted by Sun Microsystems and other participants in the HPCS program confirm that such an "expertise gap" does exist and does exert a significant confounding influence on HPC application development. Further, the nature of the "expertise gap" appears not to be amenable to previously proposed solutions such as "more education" and "more people." A productivity improvement of the scale sought by the HPCS program will require fundamental transformations in the way HPC applications are developed and maintained.

Categories and Subject Descriptors
D.2.0 [**Software Engineering**]. D.1.3 [**Programming Techniques**]: Concurrent Programming –*parallel programming*.

Keywords
High Performance Computing, Software Productivity.

1. INTRODUCTION

The development of application software in the High Performance Computing (HPC) domain is extraordinarily difficult. Indeed the Defense Advanced Research Project Agency (DARPA) has called out programmability as one of the key goals of the High Productivity Computing Systems (HPCS) program. DARPA understands productivity to be a composite of system properties and has set HPCS program goals in each [2]:

- **Performance** (time-to-solution): speedup critical national security applications by a factor of 10X to 40X
- **Programmability** (idea-to-first-solution): reduce cost and time of developing application solutions (~10X)
- **Portability** (transparency): insulate research and operational application software from system
- **Robustness** (reliability): apply all known techniques to protect against outside attacks, hardware faults, & programming errors
- A fifth property, **Systems Administration**, was added at the suggestion of Sun Microsystems, one of the HPCS vendors.

Acknowledging that productivity is fundamentally not well understood, DARPA has also funded a broad research program on HPC productivity that engages universities, research labs, the program vendors, and the HPCS "Mission Partners," including the Department of Defense, Department of Energy and federally funded sites such as Sandia National Laboratory and Los Alamos National Laboratory.

Of the system properties that make up productivity, programmability is the least well understood and perceived to be the most problematic. It has been suggested that an "expertise gap" is a significant barrier to increased software productivity: "Programming today's HEC[1] systems requires a high level of expertise, but the current trend of available skills is increasingly one of few HEC expert programmers …" [12].

A goal of the HPCS Program is to create a base of empirical data describing all aspects of productivity. Although the program is far from complete, preliminary analysis of data concerning programmability confirms that there is in fact an HPC "expertise gap" and that it is

Conference PPHEC'06, February 12, 2006, Austin, Texas, United States.

[1] High-End Computing, another term for High Performance Computing (HPC)

fundamental to the ways software is developed in many areas of the HPC domain.

This paper presents those preliminary analyses, beginning in section 2 with a discussion of the environment in which HPC software is developed, maintained, and evolves. Many of these insights about the environment have themselves been produced by HPCS program research. Section 3 describes the interdisciplinary research methodology used by the Sun productivity program, drawing on a variety of social science and empirical software engineering techniques. Selected preliminary findings are presented in section 4, with primary emphasis on suggestive case studies. Section 5 discusses some of the implications of these findings for the HPCS program goals.

2. BACKGROUND AND RESEARCH GOALS

The HPC environment is characterized by very large problems (measured by both data size and computation) that require very large, sometimes exotic computing systems and the exploitation of very high degrees of parallelism. The focus of the HPCS program has been on "Mission Partners" in the United States: the Department of Defense, the Department of Energy, and the intelligence community. HPC also takes place in commercial environments and those have been the subject of some studies as well.

Case studies published to date describe an environment that has much in common with other kinds of software development, but which differs markedly in a few respects [4][5][9][10]. The following characteristics are important for the purposes of this discussion:

1. HPC codes typically take years to develop.
2. Once developed, HPC codes have a maintenance and evolution lifetime that can be counted in decades.
3. Because HPC codes have such a long use life, they may be ported to new machines several times during their lifetime, possibly every 3-4 years.
4. HPC codes embody very complex science and numerical methods.
5. As a result of their age, legacy codes are typically written in Fortran77 and MPI. This sets them apart from other kinds of programming.
6. New code is often being written in C++.
7. HPC codes are often supported by tools that are specifically written for them.

The goal of our research is to collect empirical data and build understanding of HPC application development, with a particular emphasis on characterizing the key "bottlenecks" that appear to have negative impact on application development effectiveness. From this data we intend to develop a model for reasoning about programmability and for predicting how bottlenecks might be ameliorated.

3. METHODOLOGY

To gather more detailed information about HPC software development, we adopted a case study approach; this approach provides a data collection framework that is flexible and allows deep exploration of one or more cases [4][5][9][10]. We used multiple methods from case study research [14] that would allow us to gather information from professionals and teams of professionals who are writing code for highly parallel machines. These methods included qualitative data collection including:

- Semi-structured interviews with individual HPC programmers.
- Structured group sessions (surveys and interviews) with existing Mission Partner code teams.

We also used quantitative methods that allowed us to validate case study findings across a larger sample, including a survey of Mission Partner teams, using a multiple choice and open-ended questions format.

The use of mixed methods, measuring, observing and interviewing programmers, has a number of advantages over reliance on one type of data collection alone. For example, journal entries allowed us to collect "real-time" accounts of code development from a programmer perspective. Individual and group interviews help us understand long-term team and context issues. Survey results provide a structured overview and validated observations.

Of course the raw data alone did not provide us with the insights we seek. By weaving together the qualitative data, we were able to compare journal entries, survey responses, and interviews in order to fill out our understandings and isolate inconsistencies. From the combined data we then began to identify patterns across individuals and teams, plot bottlenecks and create models of HPC programmers based on empirical data.

4. FINDINGS

The data collected so far shed light on the suggested "expertise gap" that is widely reported in anecdotal form, for example as noted by Sarkar et. al. [12]. Analysis can be carried out from more than one perspective. This paper examines insights gained by identifying "bottlenecks:" barriers to the accomplishment of development goals. Analysis from a workflow perspective will be reported separately.

4.1 "Hot Spot" interviews

The Sun Microsystems productivity team began to investigate this evidence with a systematic analysis of data collected from interviews with five Mission Partners, conducted through DARPA's productivity research effort. There were three components to the data [3]:

1. A set of proposed workflow diagrams, describing idealized work processes in different aspects of HPC

software development; respondents were invited to review the diagrams and mark those workflow nodes they perceived to be "hot spots" causing particular difficulty or expense.

2. Free-form comments from interview respondents.
3. An oral debriefing of the interviewer, during which additional respondents' comments, not otherwise recorded, were reported.

No clear consensus emerged from the respondents' node selections, but additional analysis that included both sets of comments suggested that respondents had experienced some difficulty mapping the proposed workflows into their own work practices. A revised analysis at a coarser granularity, taking full account of the comments, revealed a clear consensus, as represented in the following chart.

	MP1	MP2	MP3	MP4	MP5
Develop HPC Code	x	x	n/a		x
Debug, Test, V&V	x	x	x	x	x
Optimize Code	x	x	x	x	x
Schedule/ Run Code	x		x		x
Math Libraries	x	x	n/a		x

Chart of Mission Partner Bottlenecks

The chart depicts the areas where data revealed each Mission Partner to be experiencing difficulty, i.e. where HPC code developers spend the greatest time and effort. Five areas were reported by a majority of Mission Partners:

1) Developing HPC Code,
2) Debugging, Testing and Verification & Validating,
3) Optimizing Code,
4) Scheduling Code Runs, and
5) Creating, Selecting, and Using Math Libraries

Areas identified by fewer Mission Partners do not appear in the chart. Mission Partner 3 does not develop code.

Importantly, of the five areas that consume time and effort, Mission Partners unanimously reported that optimizing, debugging, testing, and Verification and Validation of code were bottleneck areas.

We concluded that scaling code for HPC systems was the most likely area to become a bottleneck, which supports anecdotal reports that specialized "expertise" is crucial for scaling code for High Performance Computing (HPC). A lack or "gap" in the availability of that expertise might be an important factor blocking increased HPC productivity.

4.2 Classroom "defect" studies

Recent quantitative research from a study conducted at the University of Maryland has confirmed similar bottleneck areas while studying novice developers [7]. In this study, students were monitored while developing solutions to small HPC programming problems; all of the students' code runs were examined and the "defects" in faulty codes categorized. Those findings match the first two bottleneck areas identified in Sun's analysis: Coding for HPC, and Optimizing Code. The novices being studied do not engage in validation, scheduling or significant library use.

We realized that we would need to conduct additional research to explore and validate our initial findings as well as expand our knowledge of the expertise gap.

4.3 Scale up with more education?

When people are working hard but do not seem to be productive enough, it is tempting to think that more education might address any "expertise gap."

Will more or better education help solve the expertise gap? In the case study research with professional HPC developers, we included questions about education to help us gain a better understanding of the context of the developer's work and associated education challenges. The following example is representative of the information we collected.

Don[2] is typical of the HPC developers with whom we talked. His work history provides an example of the challenges associated with solving the expertise gap through education.

Don has a Ph.D. in Geophysics with a special interest in meteorite impact on planetary bodies. Because of his background in impact studies, he was recruited onto the Condor team at a Mission Partner Laboratory whose legacy code modeled impact [5].

The code that Don began to learn has existed for 20 years, is written in Fortran77 and contains well over 100,000 lines. Over the years this code has developed a large user base: more than 300 licenses with about 1000 users. Its users perceive this code as valuable, and Don was hired to maintain the code: fixing bugs reported by users and upgrading it when necessary.

Surprisingly, Don joined the code team with no knowledge of Fortran or MPI. He reports that it took him eight years to learn his job. Of course it did not take him eight years to learn Fortran; that took about eight months, which he called his "on-the-job training." The reason for his long learning curve is the complexity of the code itself. The code Don works on has had many authors over the years. It has also been ported to new machines several times during its lifetime use. Not all authors documented their work, and

[2] "Don" and other names that appear in case study descriptions are pseudonyms, used for confidentiality and security.

many legacy features in the code were left intact each time the code was ported.

Don eventually became the lead developer for this code. Although Don now feels confident with the code, he admitted that, after fifteen years working with the code, he still does not understand what all the code does.

Don's long learning period appears typical for HPC development. For example, Sarkar et. al. also reported a 10+ year hands-on learning curve to develop such expertise [12].

Formal education in languages such as Fortran or parallelizing tools such as MPI or OpenMP would not have helped in Don's case. He quickly mastered the skills in these areas. Instead the bulk of his learning was centered on acquiring experience with the legacy code. It was the complexity of this large code that demanded the extra time to master.

The learning curve requirements of the HPC developer working with complex code casts doubt on any long-term strategy of relying on education to solve the problem. Even if we could provide appropriate education, the timeframe for such a solution is unrealistic. The hope that the expertise bottleneck might be solved through education is not a long-term answer.

4.4 Scale up with more people?

If more education alone will not solve the expertise gap perhaps building teams of skilled people can cut down on the time to solution. In another example from our case studies we learned that the answer is "maybe."

The Hawk team filled out surveys and participated in a group interview as part of our case study research [4]. Asok was a founding member of the team, having been recruited as a graduate student. His expertise in fluid dynamics was important to a new project challenged to model a hi-tech plastic to be used in airplanes and armored vehicles, whose parts would be fabricated using large molds. However, it is difficult to fill the molds correctly, without air pockets. Asok was asked to model the mold filling as an alternative to expensive experimentation. Although Asok was knowledgeable about the science, he was not an HPC programmer, so he was teamed with a Fortran expert.

Asok understood the fluid dynamics, the programmer understood Fortran, but they did not understand each other. Four years later the Fortran code was unsuccessful, and the programmer quit.

A new programmer was assigned to work with Asok. Mitch was more fluent in C++ and suggested abandoning the existing code. It was a bold move, but the two decided to begin all over again with C++. They worked together to find a working solution to the modeling, this time creating a successful serial code. Unfortunately neither was expert in scaling. Their manager, Jean, was an expert in optimizing code; she stepped in to scale the code, working closely with Mitch.

Meanwhile the project sponsor started to become impatient, and a manager was still needed to handle expectations. So the project management was taken on by another person. His main role was to run interference for the team, keeping the sponsor happy. He also negotiated time to run the code on the large machine.

It took an additional three years to develop a successful parallel code in C++ but the team effort paid off. The new C++ code was delivered to the sponsor.

Asok attributed the success to two factors: his new programming partner and the coding knowledge he gained during the four years he spent attempting to write the code in Fortran. The C++ programmer, Mitch, attributed success to Asok's expertise in fluid dynamics and Asok's willingness to teach Mitch about it. Both agree that the project would not have succeeded if their manager had not stepped in to scale the code for a highly parallel machine.

This case study illustrates the potential for overcoming the expertise gap with appropriately configured teams. In less than seven years the Hawk team demonstrated that they could develop a working code in less time than a lone developer. But it is a cautionary tale. The team was successful only when they had the appropriate mix of knowledge represented in four areas:

- Science
- Programming
- Scaling / Optimizing
- Management

The success of a team strategy for overcoming the expertise gap relies on a conscious division of labor with a specific mix of domain knowledge. In the Hawk case, the first effort at assembling a team did not work. Throwing more people at the problem without consideration for the team skill mix led to four years of frustration. It was only when the mix of skills was balanced between science, programming, scaling, and management did the effort move forward.

It is also clear that Asok still had to acquire a working knowledge of programming in order to find a solution. He admits that the four years working with the Fortran programmer enabled him to work successfully with the C++ programmer with whom he was subsequently partnered. The programmer needed to gain a working knowledge in Asok's area of expertise as well.

This example suggests two "best practices" that can help a team to succeed.

1. The team needs the right mix of skills: science, programming, scaling/optimizing and management.
2. It is also important that each team member has a basic working knowledge of the other skill sets on the team so that they can communicate effectively.

But assembling a team of appropriate experts is not a complete solution. Of the four skill sets, scaling/optimizing remains a scarce resource, and if Jean had not possessed those skills the project might have stalled. By putting a team together that addresses the various expertise needs, the team approach can bridged the expertise gap. However this is temporary. As systems get even larger and more complex, this key expertise will become relatively more scarce, and teams will be unable to scale. For example adding more programmers to Asok's team would not have helped solve the problem any faster. In fact, more team members would have added a team / human complexity, without fundamentally addressing the "expertise gap."

5. DISCUSSION

A combination of qualitative and quantitative research methods allowed us to develop a more complete understanding of the issues faced in the HPC community than might have been possible with any single method. Using a case study approach allowed us to deeply explore the work of individuals and teams of HPC code developers. Quantitative method provided the breadth of data to validate our findings on a larger scale.

From our research we conclude that complexity is at the core of HPC bottlenecks, and it is a system level problem.

Previously proposed solutions such as "more education" or large team approaches are short-term interventions that will not be viable unless we address the root cause of the "expertise gap" – increasing complexity [6]. Very few individuals have the complete set of skills (science, programming, scaling and management) necessary to exploit fully these complex machines. Educating individual developers in all four skills requires both a gifted individual and many, many years.

Assembling a team of experts is an alternative that has short-term potential. However, these teams have a limited ability to scale. Doubling the number of scientists or programmers adds another layer of complexity to the team. As machines get bigger and more complex the pool of experts who have the ability to deal with the increasing level of complexity will continue to narrow. Unless we address the system level cause of the expertise gap, increasing complexity, we will not solve the problem.

6. CONCLUSIONS

Productivity improvement on the scale sought by the HPCS program must address scale and complexity by requiring fundamental transformations in the way HPC applications are developed and maintained [1][7][11][13]. In the end we believe that bottlenecks will be resolved with appropriate application of automation, abstraction and associated tools. Abstraction, including languages may solve the science domain expertise gap by reducing the programming complexity and allowing scientists to reason in the problem domain. Opportunity here now revolves around grounding practices in specific problem domains, automating that which can be automated, and abstracting away the most challenging aspects of the machine (parallelization).

7. ACKNOWLEDGMENTS

We are grateful to all of our HPCS program colleagues at Sun Microsystems, especially Eugene Loh, Michael Ball, and Victoria Livschitz. We also thank Doug Post, Richard Kendall, Jeremy Kepner, and many others in the HPCS community for their helpful discussions and comments.

This material is based upon work supported by DARPA under Contract No.NBCH3039002.

8. REFERENCES

[1] Ahalt, S. C, and Kelley, K. L., Blue-Collar Computing: HPC for the Rest of Us. *ClusterWorld, 2*, 11(Nov. 2004).

[2] Defense Advanced Research Project Agency (DARPA) Information Processing Technology Office, High Productivity Computing Systems (HPCS) Program. <http://www.darpa.mil/ipto/programs/hpcs/

[3] Kepner, J, Personal Communication, 2005.

[4] Kendall, R. P., Carver, J., Mark, A., Post, D., Squires, S., and Shaffer, D., Case Study of the Hawk Code Project, Los Alamos National Laboratory Report LA-UR-05-9011, 2005.

[5] Kendall, R. P., Mark, A., Post, D. E., Squires, S., Halverson, C., Case Study of the Condor Code Project, Los Alamos National Laboratory Report LA-UR-05-9291, 2005.

[6] Levesque, J., Have We Succeeded Because of Complex HPC Software or In Spite of It? Times N Systems, Inc., (August 17, 2001). <http://www.etnus.com/Company/press/press_release.php?file=hpc>.

[7] Loh, E., Van De Vanter, M. L., and Votta, L. G., Can Software Engineering Solve the HPC Problem?, *Proceedings Second International Workshop on Software Engineering for High Performance Computing System Applications*, St. Louis, 15 May 2005.

[8] Nakamura, T., University of Maryland, Personal Communication, 2005.

[9] Post, D. E. and Kendall, R. P., Software Project Management and Quality Engineering Practices for Complex, Coupled Multi-Physics, Massively Parallel Computational Simulations: Lessons Learned from ASCI, *International Journal of High Performance Computing Applications: Special Issue on HPC Productivity*, J. Kepner (ed.), **18**(4), Winter 2004.

[10] Post, D. E. Kendall, R. P., and Whitney, E. M., Case Study of the Falcon Code Project, *Proceedings Second*

International Workshop on Software Engineering for High Performance Computing System Applications, St. Louis, 15 May 2005.

[11] Post. D. E., and Votta, L. G. Computational Science Requires a New Paradigm. *Physics Today*, **58**(1): p. 35-41.

[12] Sarkar, F., Williams, C, and Ebcioglu, K, Application Development Productivity Challenges for High-End Computing, *First Workshop on Productivity and Performance in High-End Computing* (P-PHEC), Madrid Spain. February 14, 2004.

[13] Squires, S, Tichy, W. G., and Votta, L.G. What Do Programmers of Parallel Machines Need? A Survey. *Second Workshop on Productivity and Performance in High-End Computing* (P-PHEC), San Francisco, Feb. 13, 2005.

[14] Yu, R. K., *Case Study Research: Design and Methods* SAGE Publications, 2002.

The Value Derived from the Observational Component in an Integrated Methodology for the Study of HPC Programmer Productivity

Catalina Danis and Christine Halverson, IBM TJ Watson Research Center, Hawthorne, NY
{danis, krys}@us.ibm.com

1. INTRODUCTION

It is a common lament within the High Performance Computing (HPC) community that programmer productivity is not keeping pace with improvements in processor performance. This gap is in part attributable to problems of human productivity that derive from the complexity of the programming task. Consequently, understanding the work of HPC programmers is of utmost importance if the community is to make significant progress in narrowing the current gap between the performance capabilities of high-end computing (HEC) systems and programmer productivity.

This paper discusses a quasi-experimental methodology for the study of programming work which integrates an automatically collected stream of data with a manually collected stream to provide a comprehensive view of programming work. The methodology is an example of the so-called hybrid methodologies (Hochstein et al, 2005) previously described in the literature which exploits the complementarity of the two streams of data to arrive at more accurate and more comprehensive understandings of programming activities. For periods of programmer activity where there is a gap in one stream, the other covers the gaps. And, where both records are available, each can augment the other.

The scope of programmer work which is of interest to us begins with the programmer developing an understanding of the problem to be solved. While work at this stage can involve work on a computer such as reading and experimentation, this work is conceptual and frequently does not leave behind digital traces. The next segment of the programmer's work consists of developing a high-level conceptual solution for the problem. This part of the programmer's work is again focused on understanding, for example, how a proposed algorithm would work in detail. As such, this work might involve diagramming and sketching the behavior of the algorithm using informative data values. In today's programming environments, much of this work is done with paper and pencil rather than with the use of a computer (Hochstein et al, 2003). The third segment of the programmer's work is primarily focused on the use of programming resources to create an implementation and consequently leaves behind a rich record of activity carried out on the computer. A methodology that combines data streams capable of recording activities which take place both off and on the computer is therefore critical for producing a comprehensive view of programming work.

In this paper we present the discussion of the integrated methodology from the standpoint of the value provided by the manual component. A detailed explanation of the automatic component is available in Nystrom et al (2005). Our focus is on two uses of the manual data: when it is the sole record of programmer activity and when it overlaps with data collected through the automatic component. (See Figure 1.) In both cases, the value stems from the complementarity of the two data streams. When combined, they provide a more complete picture of the sequence of activities that programmers engage in to complete programming tasks.

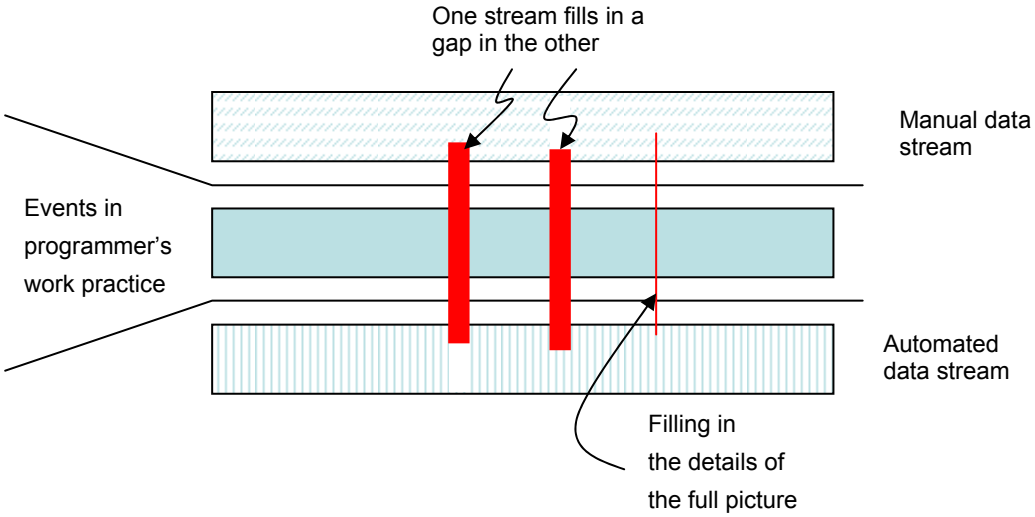

Figure 1. This diagram illustrates how both data streams–while fine grained–only capture a portion of the total events happening in the programmer's practice. By combining data from each stream we get a more detailed look at the extent of the programmers' activities.

2. BACKGROUND ON METHODOLOGIES FOR MEASURING PROGRAMMING ACTIVITY

Hochstein et al (2005) refer to methodologies that combine manually and automatically collected data as *hybrid* methodologies. In principle, manual collection efforts can produce a record of the entire programming process, from initial understanding of the problem through optimization of the solution for an HPC platform. However, the cost of collecting the data and practical limitations on the granularity of the data collected has generally resulted in limited data being collected. Furthermore, the cost of analysis is similarly high, including the transcription from analog records to digital. Automatic data collection is limited to recording computer-based events, though it can produce very detailed, consistent and error-free data (Nystrom et al REF).

2.1 Manual Observations

One widely used approach for the manual capture of data is through self reports, either through the analysis of free-form diaries kept by programmers (e.g., Perry et al, 1995) or through the direct selection by programmers of items from pre-specified lists (e.g., Hochstein et al, 2005).

Free-form self reports in which the programmer records his or her observations at any point in the programming task are easiest to collect but are prone to several problems. For example, Perry et al (1995) in a post-analysis of project notebooks and programmer diaries found large variability in the ways activities were reported and found insufficient resolution in the labels generated by the programmers; both problems complicating data analysis. These problems can be ameliorated by requiring respondents to select from pre-specified lists of activities that have been identified as informative of programmer activity.

Nevertheless, all self-reports, require programmers to log their activities and therefore cause interruptions in the primary programming task.. Switching context between programming and observing oneself programming introduces a significant overhead for the programmers and can result in inaccurate reporting (Johnson et al, 2003). Furthermore, being subjective, these assessments lend themselves to possible distortion due to a conflict of interest. For example, a student reporting on his effort in programming a classroom exercise might be tempted to either under- or over-report the time he spent on the task in order to influence the instructor's evaluation appropriately.

Additionally, if the logging activity is done after the programming task, its accuracy can be affected by human memory limitations. Forgetting is especially likely when time

intervals are filled with other activities (Wixted, 2004). However, memory problems can be decreased by prompting programmers to make self reports at regular or semantically meaningful points in the programming activity. For example, Hochstein et al (2005) requested effort assessments by programmers whenever they submitted a program to the compiler. These researchers were able to improve the accuracy of time estimates collected from self-reports (relative to reports by a human observer) by requesting that the programmer specify start and end times rather than to report time estimates only. Without specifying start and end times, the programmers appeared to be rounding off their responses to the nearest quarter hour, thus introducing errors into their estimates.

While prompted and structured self-report data increases their accuracy, there remain significant problems from the standpoint of the programmer who is asked to provide the data. One indication of this is that programmers fail to adopt the measurement technology in their work practice outside of the classroom (Johnson et al, 2003).

2.2 Automatic Observations

The above problems can be eliminated by instrumenting the programmer's workstation and collecting the data automatically. The resulting data record is objective, accurate and as detailed as the particular instrumentation allows. For example, Hochstein et al (2003) instrumented the programmer's compiler to collect data to estimate program development time. A broader assessment of programmer activities is possible through the approach embodied in the Hackystat system developed by Phillip Johnson and colleagues (Johnson et al, 2003). Hackystat consists of an extensible framework for recording programmer activity by means of "sensors" which record tool-specific programming activity data (has both server and client side components). The starting set of sensors built in to Hackystat by its designers, including support for the Emacs and the JBuilder IDEs, the ANT build tool and the JUnit testing tool, has been extended by others (e.g., the Eclipse sensor developed by Turker Keskinpala at Vanderbilt University). Not all clients can be "wrappered," that is, some do not support the attachment of sensors. Johnson et al (2003) report, for example, that the editor Emacs lends itself to this approach, while the Windows system editor—Notepad—does not.

All of the programmer actions captured from one or more sensors that instrument a workstation and its associated system output are collected in a central database to support subsequent analysis. In its original formulation, Hackystat produced a Daily Diary which combined the various data streams into an abstraction of programmer activity. Its intended use for monitoring of programmer performance is seen in its support for defining alerts that are communicated to the programmer. For example, to signal to the programmer that some threshold value based on a complexity metric has been exceeded (Johnson et al, 2003).

However, Hochstein et al (2003) have pointed out problems that result from the incompleteness of the data collected by some of the sensors. For example, while "wrappering" is possible for some editors, the measure of activity derived from such a sensor is dependent on the existence of computer events. Thus, if a programmer has an editing session open but is not producing keystrokes, perhaps because he is reading his code, the Active Time analysis will show him to be inactive. The automatic instrumentation necessarily misses events which are not computer-based. Furthermore, gaps in activity detectable through automatic means, such as those reported by Hochstein et al (2003) may contain activities that should be legitimately included in a measure of programmer on-task time. Automatic data collection systems like Hackystat (see also GRUMPS by Thomas et al, 2003) necessarily produce an incomplete record of programming activity and thus point to the need for hybrid or integrated methodologies.

3. DESCRIPTION OF THE INTEGRATED METHODOLOGY

3.1 Overview

Like the hybrid methodology of Hochstein et al (2003), the integrated methodology we report on depends on the combination of automatic and manual observations that are carried out concurrently. However, instead of depending on self-reports for the manual component of the data, our integrated methodology makes use of an independent observer to make the manual observations. While this still involves a judgment for the labeling of the programmer's activities, the use of an independent observer, eliminates two of the problems noted above with self-reported data. It eliminates both the interruption of the programmer's focal activity

due to switching context to reporting on his activity and the potential for conflict of interest in the content of the manual observations. Further, the observations we recorded manually were made at a much finer level of granularity than previously reported. Our integrated methodology also includes a significantly improved method for collecting and analyzing the automatic data component. The Standardized User Monitoring Suite (SUMS) developed by Nystrom et al (2005) collects fine-grained observations of programming activities through the broad instrumentation of a programmer's workstation. Further, it makes use of data mining techniques (Hastie et al, 2001), including statistical learning and knowledge discovery, to analyze the resultant record. Rather than specify analyses *a priori*, the SUMS data mining approach enables analyses to emerge from the data. Because a broad set of data events is captured at a fine grain, it is likely that the data required by emergent analyses will be available.

3.2 First Use of the Integrated Methodology

The first use of this methodology was in a comparative programming study comprised of novice parallel programmers (largely students) who were tasked with devising a parallel implementation of the alignment portion of Smith-Waterman algorithm (SSCA#1). Participants were assigned to one of three language groups: C + MPI, UPC or IBM's new language, X10, such that the three groups were balanced with respect to average computing experience of participants.

The study took place at the Pittsburgh Supercomputing Center (PSC) from May 23, 2005 through May 27, 2005. The three groups received training in their assigned language during the first two days of the study and then had the next two days to develop a parallel program implementing the alignment portion of SSCA #1. The assessment problem was presented at the end of day 2 so participants had an opportunity to think about the problem before they returned to the experimental environment on day 3. The three language instructors were available during the two day assessment period to answer questions. They were limited to clarifying language constructs and answering questions about the environment. They were specifically counseled to avoid helping the participants with the assessment problem. The final half day was spent on debriefing the participants about the problem and interviewing them about their experiences.

While the three languages were at different stages of maturity, we endeavored to equate the environments as nearly as possible. This meant that we did not provide debuggers, since none was available for X 10; participants were limited to using println statements for this purpose. All development occurred on a 3000-processor AlphaServer SC system at PSC (Tru64 OS, 2-rail Quadrics). X10 was run in emulation mode.

3.3 Manual Component Data Elements

We made manual observations of programmer actions by having a human observer stand behind the programmer and note down programmers' actions during the assessment period of the study (days 3 and 4 of the study). We synchronized the manual observations with the automatic observations by time-stamping the start of manual observations with respect to the clock time on the server where the automatic (SUMS) recordings were saved.

The difference in the work of the human observer in our integrated methodology and that which was reported by Hochstein et al (2005) has to do with the nature and the granularity of the observations made by the observer. Hochstein et al (2005) decided to use a human observer after they became suspicious of the validity of the data they had collected through self-reports and through instrumenting the compiler which was used by the programmers in their study. They ran two pilot studies in which a human observer sat by the programmer and produced a log of his activities by selecting from the same pre-specified activity categories as used in the self-reports. These included such high-level events such as understanding the problem, experimenting with the environment, parallelizing and testing. In contrast, the human observers in our integrated methodology noted behaviors at a much lower level of granularity, capturing events such as opening a file, issuing a batch command, inserting new code in a text file or non-computer based events such as when the programmer removed his hands from the keyboard or talked to the instructor.

A second major difference is that the human observer in our study sampled the activities of the programmers for five minute periods several times over the course of a day rather than observing a single programmer for the entire period of the study. This enabled us to cover 27

programmers with three human observers over the course of two full days of assessment.

This sampling method, while not providing the ideal of complete coverage of all participants at all times, is well validated in decades of animal behavior research and has often been used in human group observations. (E.g. see work by Shirley C Strum on studying baboon troops in Africa).

The focus of our manual observations was on all of the programmer's activities, whether on or off the computer. Since this was the first use of the integrated methodology, our initial strategy was to capture as much detail as we could. We will refine both the amount and the granularity of the observations and the types of events we record based on subsequent analysis of the data.

Prior to carrying out the manual observations, we specified the events that were of interest to us and assigned codes to these to make it make it more efficient for the observers to note down the observations. We aimed to capture events at semantically meaningful levels where possible, rather than to record literal transcription of keyboard events. For example, we distinguished between editing existing code (i.e., modifying it) and writing new code, but did not note down the content entered by the programmer. Capturing semantically meaningful descriptions is important because it can, for example, help with distinguishing between both writing and debugging code, and styles such as cut and past. Both of these are of relevance to understanding where effort is spent during programming. We also recorded shell commands, cursor movements within files and changes in window focus.

Sample non-computer events we captured include talking to the instructor, diagramming the solution and reading paper-based documentation. These are activities that are generally recognized as providing important support to programmers. In addition, we also captured low-level, non-verbal events such as removing one's hands from the keyboard, cracking one's knuckles and "dancing" in one's seat (the use of headphones and music players was common among the study participants). All of these may or may not be important for understanding programmer success, however they are evidence of how programmer time is being used. Appendix A lists the final set of codes we used to record our manual observations and we describe how we came up with them in the following section.

3.3.1 Manual component collection method

We employed three observers (the two authors of this paper plus a second year Computer Science graduate student) to cover the 27 participants who were expected to take part in our study (in actuality, 26 started the study and 25 completed it). After agreeing to a first set of codes through informal discussion, we carried out joint observations of several individuals during one of their learning practice periods and resolved differences in our coding practice through subsequent discussion. We repeated this 3 times until we came up with a precursor to the list in Appendix A. The observer selected from the pre-specified set of codes where possible, and entered unanticipated behaviors in free form. We coded these afterwards based on post-observational discussion, resulting in the final list. (Some free form comments remained in the transcriptions.)

We also used this learning period to refine our sampling technique. Manual observation carried out by human observers in real time is expensive. Since we had to plan for 27 participants in our study, and had access to only three observers, we had to devise a sampling methodology to provide even coverage of the programmers' activities. Drawing on observation techniques from animal behavior (where one person is often observing a group of animal interactions) we chose a sampling method that would allow us to spread ourselves across the full subject pool. (Lehner, 1996) Each observer was assigned a group of nine participants per observation period. Within each participant group of nine programmers the observer made observations on an individual for five minutes and then moved on to the next member of his or her group. This resulted in each programmer being observed for two five minute periods, separated by 45 minutes, during each 1.5 hour observation period.

In the next observation period, we shifted the assignment of observers to study a different set of participants. Over the course of a day, each participant was observed for at least two five minute periods by each of the three observers. In this way, we were able to eliminate any systematic effect of observer bias from our study. Individual differences in observer background (e.g., computing expertise) and in personal characteristics (e.g., quality of eyesight)

necessarily impact observer performance. Structuring the observation protocol in this way controls for any remaining observer bias not eliminated through training.

There were five observation periods on the first day and four on the second day. In total we collected 33 hours and 30 minutes of manual observations. The number of observations per five minutes ranged from 0 to 52, with an average of 27.4.

We began each five minute observation period by noting down the starting state of the participant's screen. We noted any open windows, where possible, their content (e.g., editor session, shell) and the window which had focus. We provided participants with notepads for their use during the study period. We collected these at the end of each day and re-distributed them the next morning.

3.4 Automatic Component Data Elements

In this section, we list and briefly describe the data acquisition components that comprised the automatic component or the SUMS part of the integrated methodology at the time of the study. For a more in depth treatment of SUMS, see Nystrom et al (2005). Data acquisition components were implemented through a variety of means, including program wrappers, cron scripts, C utilities and third-party software. The instrumentation enabled the collection of the following event types.

1. sums_source captures various statistics to describe the state of source files at consecutive two-minute time slices, including the total lines of code contained, the number added, deleted and changed since the previous recording.

2. sums_web keeps track of the time-stamped access to URLs with enough identifying information to be able to distinguish between access to web-based documentation and access to URLs which are not relevant to the development task (e.g., doing email, playing games).

3.sums_shell records all commands executed from a shell.

4. sums_window captures the name and associated information for the active window.

5. sums_compiler captures each invocation of the compiler and the associated exit status, errors, warnings and other output.

6.sums_batch records output from the batch commands used to invoke the AlphaServer machine.

3.5 Automatic component analyses

The approach used in the SUMS tool is to exhaustively instrument machines used in the development of scalable code, to aggregate the data collected over time, tasks, populations and so on. and then to statistically correlate system features to productivity. A major early thrust of the analyses was to assign programmer actions recorded with the SUMS system to code development phases (authoring, debugging, parallelizing, executing, other). To accomplish this, development was first quantized into uniform intervals. Then, for each interval, inputs from all of the automatic data acquisition components were heuristically integrated to assign the interval to one or more development activities. Development times for the various phases are calculated by summing over the weighted intervals. In addition, first parallel solutions were identified for each participant through a combination of automatic analysis of compiler logs and program output, and manual examination by the instructors. Based on learning from the instructors' processes, the automatic analysis to identify first solutions was refined.

4. Results based on Analyses of Data from the Manual Component

Analyses of Manual Observations

Analysis began with transcribing the handwritten codes into a digital form that then could be more easily integrated to observation streams per person. Manual observations were time-stamped (at a one minute resolution) and synchronized with the clock on the SUMS server machine. The manual data stream produced a dense stream of observations, including both on and off-computer actions, captured with a high degree of detail. Figure 2 illustrates a typical transcript of the manual observations stream.

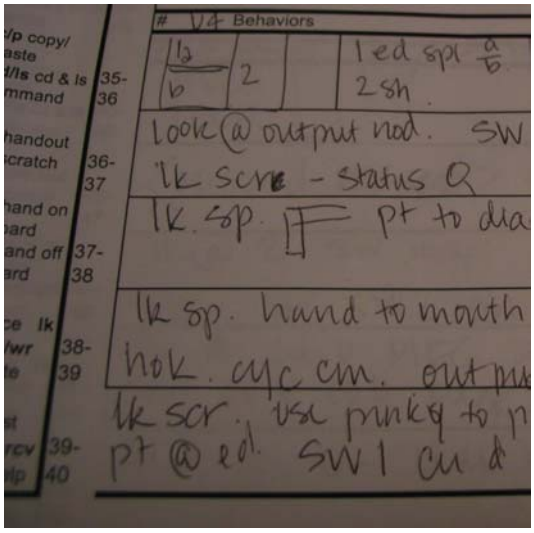

Figure 2. Snapshot of hand written notes taken during observation and then transcribed.

The amount of detail captured is much higher than has been reported by other researchers making manual observations, whether collected through self reports or through human observers (e.g., Hochstein et al, 2005). As was noted above, the median number of observed events in a five minute period was 27.

When integrated, the two data streams provide a more complete record of the programmer's activity than either does alone. In some cases, gaps in one type of record are filled-in by data from the other. In other cases, the two records overlap in time, but provide somewhat different, complementary pictures of programmer activity due to differences in the type and granularity of observations collected.

When the manual stream is integrated with the automatic stream, the manual observations contribute to two classes of analyses. First, they can fill in the gaps that exist in the automatically collected stream. These may be of moderate length, as in the second example we discuss below (see Table 1 below) which occurred during the first day of coding. Or, especially during the earlier, pre-coding phases of a programming task, they can be quite long, especially for complex problems. Even for a much simpler programming task than the alignment portion of the Smith-Waterman algorithm used in our study, Hochstein et al (2005) report an example of a ten minute gap in the automatic record during which the study participant was working out a solution on paper and which they assigned to "thinking" time.

The second class of complementary analyses where manual observations provide value occurs where both streams provide data during a particular time interval. The function of the manually collected data is both to provide a more richly nuanced picture of the programmer's activity and to raise hypotheses about the meaning of data and to validate some of the inferences made based on the automatic component. For example, since the instrumentation of the editors at this point collects summary statistics rather than capturing the sequence of events the programmer follows, it is not easy to determine whether a programmer is "writing new code" by cutting and pasting from multiple files or debugging code he had written from scratch. However, this is relatively easy for a trained human observer to do. The value of the human observer is that they can generate hypotheses based on their observations and these can then be explored in the automatically collected data at a large scale.

In what follows we present two concrete examples that illustrate some of the advantages of integrating data from two streams.

Example 1: Gap analysis of within and between subject observation of work patterns

While analysis within a single subject stream provides many different patterns, the most interesting are probably the gaps that show up in the automatic data collected because the user is not interacting with the computer directly. We can observe and analyze these gaps in a number of different ways. For example, take the following observation:

Subject: M2 Date: 5/25 Coder: CH
Screen State: 1 ED (ACT); 2 ?; 3 BR
13:30 Lk scr, Lk sp, Matrix draw, Gl rev prob. Stmt., Lk scr, Gl np, Lk scr
13:31 Hok, No type, Lk scr, Cu u, Cu u, MEC, Cu d
13:32 Cu u, Lk scr, Cu u, Cu d, Cu u, Cu u, c/ Mv cu/p
13:33 Lk scr, Hok, No type, Hnk, Gesture LH, Hand to head, Hok, Mv cu, MEC, Gl prob amend., MEC
13:34 Mv cu, Lk scr + + +, Gesture rt hand, Lk sp,

Figure 3 Transcript of five minute observation period for participant 2 in the C + MPI condition.

This transcribed coding equates to the following description. Participant M2 is the second member of the C + MPI language group. He has three windows open, including an edit session

and a web browser. Starting at 13:30, M2 performed a number of actions, none of which involved the keyboard. He (in order) looked at the screen, looked into space, drew a matrix structure on the scratch pad, glanced at the revised problem statement, looked back at the screen, glanced at the note pad, and looked at the screen. In the next segment (13:31) he places his hands on the keyboard, but doesn't type, looks at the screen and then moves the cursor up through the code and makes a change to existing code (MEC) then cursors down. In the third segment he additionally does a copy from one piece of code, and pastes it to another area of code (c/ MV cu / p). In the fourth and fifth segments we see him gesturing in space with his hands in addition to making code changes and looking at the screen and the revised problem.

Comparing this entry to other observations of the same programmer helps us draw a picture of his style. He makes notes on the pad, but they tend to be quick drawings rather than working out a problem in detail. We see that he gestures with his hands, although not pointing directly to the screen. And we confirm the pattern seen here that he tends to a style of amending existing code—either by making changes or copying and pasting a code segment to another place. In contrast, other programmers observed with such a break from the computer leave the room to go on a cigarette break, or to answer a call on their cell phone. In the former case, they may still be thinking through the problem, while in the later case it is much less likely.

By comparing against other programmers we begin to get an idea that some programmers spend more time off the keyboard than others, and what they do off the keyboard is different. Some look into space and appear to be thinking through the problem while others work through small sample problems in great detail on the note pad. This included diagramming the matrix necessary to solve the problem, and "walking through it" in simulation of the algorithm – often to verify that a particular approach would work.

Why does this matter? It is important for several reasons. First because it gives us a glimpse into the cognitive aspects of conceptualizing the problem, some aspects of which may transcend novice subjects. (We would have expected to have observed more of this had we not given out the assessment problem at the end of the second day and then given them overnight to think about it prior to resuming our observations.) We also see where development time is spent and how that is affected by particular patterns of work. For example, if someone frequently works through problems off the computer then development time must include these gaps. However, if they are getting up to call someone or have a cigarette every hour, then interpretation is more ambiguous.

Example 2: Integrating analysis with automatic data recording

While these insights from a single type of data are interesting this becomes more powerful when we begin to compare across data streams. This comparison can be with other kinds of qualitative data. Here however, we want to focus on comparisons with the automatically collected SUMS observations. To see how informative comparing these analyses can be, let's look at another programmer, working with UPC and designated U6.

The PSC team analyzed the automatic data for gaps in activity. Two of those gaps are show in Table 1 below.

U6	1370	2005-05-25 09:23:50	2005-05-25 09:46:40
U6	788	2005-05-25 09:47:14	2005-05-25 10:00:22

Table 1: Two gaps in the automatically collected data stream for participant 6 in the UPC language group.

Our sampling schedule meant that we did not necessarily overlap all of the gaps that the SUMS analysis identified. But with U6 we overlap the end of one gap and the beginning of another.

Subject: U6 Date: 5/25 Coder: JW
State*1: ED
9:45 LK SCR, PG U/D, WR SP, LK HO (Global alignment)
9:46 LK SCR, HNK, HOK, CL 1, LS [quitting the editor took him back to a shell prompt], RUN, ED (Output.txt), HNK
9:47 GL SP, HOK, O (Edmiston_final.c), PG U/D, CUR D, Left hand on mouth, POINT (MIN statement)
9:48 HNK, LK HO, FLIP (glob align table), LK SCR, HNK, POINT (code)

9:49 Pen on SP, LK SCR, WR SP (in a table, "seq 1"), LK SCR, WR SP

Figure 4: Transcript of five minute observation period for participant 6 in the UPC condition.

It is evident that the two data streams are at a different level of granularity. For example in the manually collected observation stream (Figure 4 above) we see at 9:46 that the third activity recorded is placing hands on the keyboard (HOK), followed by closing his editor window, and creating a new editor window with output.txt and taking his hands off the keyboard. From the detailed information collected through SUMS we can see that all of the computer interaction happened in the last 20 seconds of the minute, meaning that U6 looked at the screen for the first 40 seconds.

Similarly at 9:47 we see that U6's interaction with the computer must have occurred within the first 14 seconds. This does raise an interesting question about what information is captured. We can take a closer look at the data streams for some more detail.

U6	2005-05-25 09:46:40	Shell	command: a.out
U6	2005-05-25 09:46:49	Shell	command: vim
U6	2005-05-25 09:47:14	Shell	command: vim

Table 2: Sequence of shell commands executed by U6 as captured by SUMS.

Here we can confirm that there is another difference in the granularity of the observations. While the automated data is more accurate in time, the human observer can pickup some data that would be difficult to instrument. So comparing Table 2 to Figure 4 we see the automatic data captures the precise time and the command executed.

U6	2005-05-25 09:46:49	shell	command: vim

Table 3: Record of shell command executed by U6 as captured by SUMS.

The entry in Table 3 equates to the manual observation in Figure 4 at 9:47: *O* (Edmiston_final.c). However the human observer is capturing other information, such as moving within the text of the file (paging up and down, and then more fine movement of the cursor) as well as actions happening off the computer during the gaps – in this case looking at the global alignment table in the handout and working out a table on the scratch pad. As we argued above, being able to see in detail what the programmer is doing while in an edit session is potentially useful for disambiguating among several phases of code development. In addition, when developing programmer's tools such low level detail gives us insight into where the programmer is expending effort.

Lessons Learned: A Critique of the Collection of Manual Stream of Data by Human Observers

While we hope that we have demonstrated the values that a human observer can bring to the collection of manual data, the method as we have used it has some drawbacks. First of all, this is a very resource intensive method. With three observers working throughout the study, we able to only collect approximately 1/9 of the total data produced. Second, the work is difficult and error-prone—even with trained observers. Various factors impact correctness of the record that is made, including the computing expertise of the observer, her ability to see the screen well (these interact), and the rate at which the programmer produces events. Many of the programmers split their work over multiple windows and in certain stages of the work, for example, debugging, moved across the multiple windows very quickly. This makes it harder for a human observer to note down precisely. Another problem is the impact the human observer produced on the programmer. In most cases, as the study progressed, the programmers seemed to take little note of us. However, in some cases we formed the impression that some events were being generated for our benefit. One participant in particular seemed to be trying to determine just how fast he needed to cycle across windows before we could not keep up.

One solution to these problems would be to "videotape" each participant throughout the study. This would produce a completely error free record. Transcription would have to be selective, however, due to the time cost. (One of the authors has found 11 to 1 to be a good rule of thumb. That is, 11 hours of meticulous transcription for one hour of video.) First, gaps in the automatically collected stream could be identified and the corresponding video segments could be coded in detail to complete the record that was collected automatically. Second, a high-level analysis of the entire tape could be done to generate hypotheses about what style of

working the programmer is using, what types of problems they are encountering, and other high level impressions that can then be related to the automatically collected data stream.

There are of course, other options. Rather than videotaping the screen content, one could directly capture screen events using screen recorder technology. For examples see Kou and Johnson (2005) and the VNC2SWF screen recorder.

ACKNOWLEDGEMENTS

This work has been supported in part by the Defence Advanced Research Projects Agency (DARPA) under contract No. NBCH30390004. We acknowledge the contributions of our PSC colleagues, especially Nick Nystrom, John Urbanic and members of the SUMS support team who were our partners in the programming study on which this paper is based. We also thank the teachers in the three language groups, Francois Cantonent, Kemal Ebcioglu, Tarek ElGazawi, Vijay Saraswat, Vivek Sarkar, and John Urbanic. Most of all, we are grateful to the 26 study participants who endured our observations with grace and good humor.

REFERENCES

1. Hochstein, L., Basili, V. R., Zelkowitz, M.V., Hollingsworth, J.K., and Carver, J. Combining Self-reported and Automatic data to Improve Programming Effort Measurement. ESEC-FSE'05, September 5-9, 2005, Lisbon, Portugal.

2. Perry, D. E., Staudenmayer, N.A., and Votta, L.G. Understanding and improving time usage in software development. In Trends in Software: Software Process, Vol. 5., John Wiley and Sons, 1995.

3. Johnson, P.M., Kou, H., Augustin, J., Chan, C., Moore, C., Miglani, J., Zhen, S. and Doane, W.E.J. Beyond the Personal Software Process: Metrics Collection and Analysis for the Differently Disciplined. ICSE, 2003, pp.641-646.

4. Lehner, Phillip N. Handbook of Ethological Methods. Cambridge University Press, Cambridge UK, 1996.

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

6. Hastie, T., Tibsshirani, R., and Friedman, J. The Elements of Statistical Learning: Data Mining, Inference and Prediction. Springer-Verlag, New York, 2001.

7. Halverson, C. and Nystrom, N.A. Human Productivity in High-end Computing. Manuscript, 2005.

8. Thomas, R., Kennedy, G.E., Draper, S., Mancy, R., Crease, M., Evans, H. and Gray, P. Generic Usage Monitoring of Programming Students. ASCILITE 2003.

9. Wixted, J. T. The Psychology and Neuroscience of Forgetting. In *The Annual review of Psychology*, V. 55, pp. 235-269, 2004.

10. Kou, H. and Johnson, P.M. Automated Recognition of Low-level Process: A Pilot Validation

Study of Zorro for Test-driven Development. csdl.ics.hawaii.edu/techreports/06-02/06-02.pdf.

11. VNC2SWF Screen Recorder. http://www.unixuser.org/~euske/vnc2swf/. Accessed February 10, 2006.

APPENDIX A: List of Pre-specified Codes Utilized by Human Observers

Objects:

SH	shell
BR	web browser
ED	editor
HO	handout
SP	scratchpad
SCR	screen

Actions:

O open (O ED == open editor, O BR == open browser)

CL close (CL ED == close editor, CL BR == close browser)

MAX	maximize window
MIN	minimize window
MOVE	move window

RESIZE resize window

SW switch to window or task (like SW HO, switch to handout)

LS an ‘ls’ command in the shell

CD a ‘cd’ command in the shell

CD/LS a series of ‘cd’ and ‘ls’ commands

CP a ‘cp’ command in the shell

CMD some other unix command

COMPILE some command sequence which results in them compiling their program

RUN some command sequence which results in them running their program

ANC add new code

MEC modify existing code

DEL delete code / comments

PR print statement

SAVE file save

EDIT make an edit (e.g., to a CMD)

FIND find text (e.g., search)

CUR M cursor movement (move, typically left and right)

CUR U cursor movement (up by some unspecified amount)

CUR D cursor movement (down by some unspecified amount)

CUR U/D cursor movement (up and down)

PG U/D page up/down

SC U/D scroll up/down

SEL select text

COPY copy

PASTE paste

C/P copy/paste

HOK hands on keyboard

HNK hands off keyboard (hands not on keyboard)

GL glance

LK look (longer than a glance)

RD read

WR write

FLIP flip through pages of a handout or scratchpad

POINT pointing to something (e.g., POINT SCR means ‘points to screen’)

RQ HELP request for help

RCV HELP receive help

VOCAL vocalization

SUBVOCAL sub-vocalization

Parallelization of a Molecular Modeling Application: Programmability Comparison Between OpenMP and MPI*

Russell Brown
Sun Microsystems, Inc.
15 Network Circle
Menlo Park, CA 94025
russ.brown@sun.com

Ilya Sharapov
Sun Microsystems, Inc.
16 Network Circle
Menlo Park, CA 94025
ilya.sharapov@sun.com

ABSTRACT

Important components of molecular modeling applications are estimation and minimization of the internal energy of a molecule. For macromolecules such as proteins and amino acids, energy estimation is performed using empirical equations known as force fields, which can be rather complicated. For example, the interactions between a protein and surrounding water molecules may be modeled using the generalized Born solvation model that requires $O\left(n^3\right)$ computational complexity for evaluation.

Fortunately, many force-field calculations are amenable to parallel execution. This paper describes the steps that were required to transform the Born calculation from a serial program into a parallel program suitable for parallel execution in both the OpenMP and MPI environments. Measurements of the parallel performance on a symmetric multiprocessor reveal that the Born calculation scales well for up to 64 processors. Scalability is roughly equivalent for the OpenMP and MPI implementations, but the OpenMP implementation performs better and requires less programming effort than does the MPI implementation.

1. INTRODUCTION

Molecular modeling is one of the most demanding areas of scientific computing today. Although the high computational requirements of molecular simulations can produce long computation times, parallel execution may used to increase the size of molecules that can be analyzed in manageable time.

Several software packages exist for molecular modeling and estimation of the internal energy of molecules using non-quantum mechanical, or empirical equations known as *force fields*. Some of the more widely known packages from academe are AMBER [26], CEDAR [8], CHARMM [6], and GROMOS [25]. A newer, open-source package related to AMBER is Nucleic Acid Builder or NAB[17], which we use as the basis for our analysis in this work. All of these packages use a similar approach to the estimation of internal energy via a set of potential functions [4].

A typical force field includes several energy terms. The most complicated energy term models the interactions between a biomolecule and the solvent, or surrounding water molecules. This energy is known as the Born free energy of solvation. Using a method known as the generalized Born [3, 10, 23] approximation, the electrostatic contribution to this energy is computed as the sum of pairwise interactions:

$$E_{\text{Born}} = -\frac{1}{2}\sum_i \sum_{j>i} q_i q_j \left[1 - \frac{e^{-\kappa\sqrt{d_{ij}^2 + R_i R_j e^{\frac{-d_{ij}^2}{4R_i R_j}}}}}{\epsilon_w}\right] \quad (1)$$

In this equation, d_{ij} represents the distance between atoms i and j, κ represents a Debye-Huckel screening constant [22] and ϵ_w represents the dielectric constant of water. R_i represents the *effective Born radius* that is a measure of the amount by which the atom i is screened from the solvent by all of the surrounding atoms k. The effective Born radius is calculated as the sum of functions of the distance between atom i and all of the surrounding atoms k [16]:

$$R_i^{-1} = \frac{1}{\rho_i} + \sum_{k\neq i} f\left(d_{ik}, \rho_i, \rho_k\right) \quad (2)$$

In this equation, ρ_i and ρ_k represent the (constant) intrinsic radii of atoms i and k, and $f\left(\right)$ is a smooth function of the interatomic distance and the intrinsic radii [19, 16].

Because of the presence of R_i and R_j in equation 1, computation of the Born free energy and its first and second derivatives can involve considerable complexity. However, it is possible to reduce this computational complexity via precomputation, the details of which are beyond the scope of this paper and are reported elsewhere [7, 21, 24]. This precomputation produces two vectors, R and A, each of length n (where n is the number of atoms). These vectors are used to produce four matrices: $\mathbf{N}$ (of size n by n) and $\mathbf{F}$, $\mathbf{G}$ and $\mathbf{D}$ (each of size $3n$ by n). Products of these matrices are used to form the Hessian matrix $\mathbf{H}$ (of size $3n$ by $3n$) that contains the second derivatives. This precomputation reduces the computational complexity of the Born free energy and its first derivatives to $O\left(n^2\right)$, as well as reducing the complexity of the second derivatives to $O\left(n^3\right)$.

For molecular modeling applications, it is desirable not only to estimate the internal energy of a molecule, but also to minimize that energy. Minimization may be performed using the Newton-Raphson [4, 18] method. One iteration of Newton-Raphson is calculated as:

$$x_1 = x_0 - \mathbf{H}^{-1}\left(x_0\right)\nabla E\left(x_0\right) \quad (3)$$

In the above equation, x_0 represents the initial Cartesian coordinates of the atomic nuclei prior to the Newton-Raphson

*This material is based upon work supported by DARPA under Contract No. NBCH3039002.

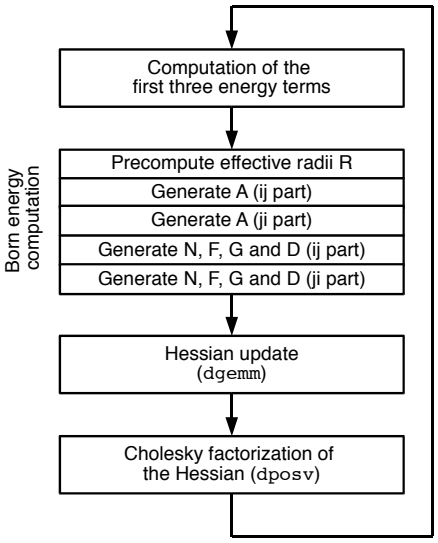

Figure 1: Phases 1-4 of Newton-Raphson. The ij and ji parts are computed by `for` loop nests.

iteration, x_1 represents the Cartesian coordinates after the Newton-Raphson iteration, $\nabla E(x_0)$ represents the gradient vector of first derivatives that are calculated from the initial Cartesian coordinates, and $\mathbf{H}^{-1}(x_0)$ represents the inverse of the Hessian matrix of second derivatives that are calculated from the initial Cartesian coordinates. In practice, inversion of the Hessian matrix is computationally expensive and is avoided by solving the linear system via techniques such as Cholesky factorization [18].

2. IMPLEMENTATION

As shown in Figure 1, each iteration of Newton-Raphson minimization is subdivided into four phases of computation: (1) calculation of the first three energy terms (the non-Born terms) as well as their derivatives; (2) calculation of the Born free energy, its first derivatives, and the matrices **N**, **F**, **G** and **D**; (3) matrix multiplication to produce the Hessian matrix **H** of second derivatives; and (4) solution of the linear system via Cholesky factorization.

We can visualize the phases of an iteration by monitoring the changes in low-level activity in the system. Figure 2 gives an example of the low-level activity that illustrates the change in the *cycle per instruction* (CPI) measurements for different phases of execution of the MPI version of NAB. The CPI measures the efficiency of the CPU; low CPI values indicate good performance. Superscalar processors are capable of executing multiple instructions in one cycle and therefore CPI values can be less than one for carefully tuned sections of code. In our experiments we measured the CPI and other low-level statistics, such as cache miss rates, using UltraSPARC® processor on-chip hardware counters [11].

In Phase 1 we observe fairly poor processor efficiency, which can be explained by non-contiguous memory references in the computation of energy terms that involve chemically bonded atoms. Relatively little computation is performed in this phase; therefore, this phase doesn't significantly impact the overall performance.

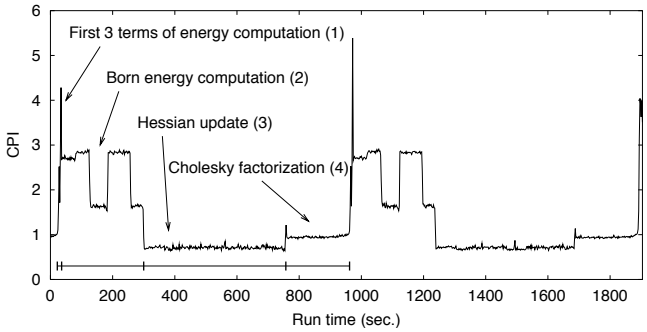

Figure 2: CPI profile for Phases 1-4 of two iterations of Newton-Raphson minimization.

In Phase 2 we can see five distinct regions corresponding to the five Born energy computations outlined in Figure 1.

Phase 3 updates the Hessian matrix via three matrix-matrix multiplications that are implemented using the `pdgemm` subroutine of the Scalable LAPACK library [5]. On a Sun™ symmetric multiprocessor, this subroutine ultimately calls the `dgemm` subroutine from the Sun Performance Library™, which is optimized for the SPARC® processor family [1].

Phase 4 performs Cholesky factorization using the `pdposv` subroutine from the Scalable LAPACK library.

To summarize, the computation in Phase 1 has $O(n)$ complexity and contributes only minimally to the total computation. The computational complexity of Phases 3 and 4 is $O(n^3)$ and is performed by parallelized subroutines from a scientific library. The computation in Phase 2 exhibits $O(n^2)$ complexity and must be parallelized in order that the total computation achieve reasonable scalability [2]. Our efforts have been directed towards the parallelization of this Phase 2 computation for two different implementations of NAB. The first implementation is parallelized via OpenMP [9] and the second is parallelized via MPI [12, 13, 15].

2.1 OpenMP

The computation of Phase 2 of Newton-Raphson minimization involves several summations such as $\sum_i \sum_{j>i}$ or $\sum_i \sum_{j\neq i}$, each of which implies a loop nest with i and j as the outer and inner loop indices, respectively. Each loop nest updates a vector and a matrix, as shown for the shared vector A and the shared matrix **N** in the following fragment of C code that is (incorrectly) parallelized via OpenMP by adding a `#pragma omp parallel for` directive to the serial code:

```
#pragma omp parallel for private(j)
for (i = 0; i < n; i++) {
  for (j = i+1; j < n; j++) {
    A[i] += f1(i, j);
    A[j] += f2(i, j); //Incorrect!
    N[i][i] += f3(i, j);
    N[i][j] += f4(i, j);
    N[j][j] += f5(i, j); //Incorrect!
    N[j][i] += f6(i, j);
  }
}
```
(4)

The above code fragment exhibits a race condition for the update of $A[j]$ and $\mathbf{N}[j][j]$. Because the j loop index

is not partitioned amongst the OpenMP threads, all of the threads can potentially update $A[j]$ and $\mathbf{N}[j][j]$. There is no guarantee that these updates will occur atomically, and therefore the threads may overwrite one another's updates. This race condition is easily removed by splitting the loop nest to create two loop nests. The first loop nest preserves i as the outer index and j as the inner index, and updates $A[i]$, $\mathbf{N}[i][i]$ and $\mathbf{N}[i][j]$:

```
#pragma omp parallel for private(j)
for (i = 0; i < n; i++) {
  for (j = i+1; j < n; j++) {
    A[i] += f1(i, j);
    N[i][i] += f3(i, j);
    N[i][j] += f4(i, j);
  }
}
```

(5)

The second loop nest uses j as the outer index and i as the inner index, and updates $A[j]$, $\mathbf{N}[j][j]$ and $\mathbf{N}[j][i]$:

```
#pragma omp parallel for private(i)
for (j = 0; j < n; j++) {
  for (i = 0; i < j; i++) {
    A[j] += f2(i, j);
    N[j][j] += f5(i, j);
    N[j][i] += f6(i, j);
  }
}
```

(6)

The combination of i and j loop indices is identical for both loop nests, *i.e.*, for either loop nest a given value of i is combined with the same values of j. It is essential that $A[i]$ and $\mathbf{N}[i][i]$ be updated in the first loop nest, and that $A[j]$ and $\mathbf{N}[j][j]$ be updated in the second loop nest; otherwise, a race condition will result. However, $\mathbf{N}[i][j]$ and $\mathbf{N}[j][i]$ may be updated in either loop nest. No race condition can exist for these updates because each update involves both i and j matrix addresses.

Nevertheless, the decision to update $\mathbf{N}[i][j]$ in the first loop nest, and to update $\mathbf{N}[j][i]$ in the second loop nest, is not arbitrary. By updating $\mathbf{N}[i][i]$ and $\mathbf{N}[i][j]$ in the first loop nest, and by updating $\mathbf{N}[j][j]$ and $\mathbf{N}[j][i]$ in the second loop nest, the matrix elements are partitioned amongst the OpenMP threads according to matrix row. Groups of r contiguous rows may be partitioned amongst the threads by adding a `schedule(static, r)` clause to the `#pragma omp parallel for` directive. This partitioning is known as *row cyclic* partitioning. This approach promotes locality of memory access by each OpenMP thread for a matrix that is allocated in row-major order. We will return to partitioning when we discuss parallelization via MPI.

Code fragments 5 and 6 exemplify the parallelization that is accomplished via OpenMP for Phase 2 of Newton-Raphson iteration. In addition to splitting some `for` loops, the parallelization of Phase 2 requires the resolution of issues such as false sharing, which occur commonly in OpenMP programs and which are beyond the scope of this discussion.

For Phases 3 (matrix multiplication) and 4 (Cholesky factorization), parallel execution is performed by subroutines from a parallelized scientific library. For example, the Sun Performance Library provides multi-threaded versions of the LAPACK subroutines `dgemm` and `dposv`, which perform matrix multiplication and Cholesky factorization, respectively.

2.2 MPI

The parallelization of Phases 2-4 of Newton-Raphson minimization is more complex for MPI than for OpenMP. The increased complexity is due principally to the Scalable LAPACK (or ScaLAPACK) scientific library that is used with the MPI implementation of NAB. The ScaLAPACK library does not support global, shared vectors and matrices; instead, it supports vectors and matrices that are distributed across all of the MPI processes. Under this distributed paradigm, each process has exclusive access to a unique subset of the global vector or matrix, which we will call the *sub-vector* or *sub-matrix*. Each process initializes its sub-vector or sub-matrix, and then the ScaLAPACK subroutines distribute computation such as matrix multiplication and Cholesky factorization across all of the processes.

Before a vector or matrix can be processed by a ScaLAPACK subroutine, it must be distributed onto a *process grid*. The process grid is a group of MPI processes that are placed on a rectangular grid of `nprow` rows by `npcol` columns. Each process has unique row and column coordinates `myrow` and `mycol` that indicate the location of the process on the grid. The matrix elements are not mapped onto the process grid in a contiguous manner but rather in a *block cyclic* manner [5] wherein a matrix of m rows by n columns is subdivided into blocks of `mb` rows by `nb` columns (see details below). Block cyclic mapping is used by ScaLAPACK to achieve reasonable load balancing across the MPI processes. For example, the solution of a system of linear equations is accomplished by sweeping through a matrix from upper left to lower right in such a way that an anti-diagonal processing wavefront advances along the principal diagonal of the matrix. Block cyclic mapping ensures that the wavefront encounters data from all processes at each stage of its advance.

Figure 3 illustrates the block cyclic distribution of a 13 by 8 matrix onto a 3 by 2 process grid, using 2 by 3 blocks. Scanning from top to bottom along the leftmost column of the matrix, we see that the upper left block of the matrix (which begins with element $a_{1,1}$) maps to the upper left block of process (0,0). The block that begins with element $a_{3,1}$ maps to the upper left block of process (1,0). The block that begins with element $a_{5,1}$ maps to the upper left block of process (2,0). The block that begins with element $a_{7,1}$ begins a new cycle and maps to the block beneath the upper left block of process (0,0). The block that begins with element $a_{9,1}$ maps to the block beneath the upper left block of process (1,0). The block that begins with element $a_{11,1}$ maps to the block beneath the upper left block of process (2,0). The (partial) block that begins with element $a_{13,1}$ begins a new cycle and maps to the lower left block of process (0,0). A similar cyclic mapping in the horizontal direction fills the process grid and produces the required block cyclic distribution of the matrix onto the process grid.

As mentioned above, each MPI process has exclusive access to a unique sub-vector or sub-matrix. Thus, the i and j loop indices of a loop nest must be restricted to only those values that are required to update the accessible sub-vector or sub-matrix elements. Moreover, the loop nest must be split because a process that can access $\mathbf{N}[i][j]$ cannot necessarily access $\mathbf{N}[j][i]$. The (partially correct) code fragment 7 satisfies these constraints and accomplishes the necessary block cyclic partitioning of matrix elements amongst the MPI processes.

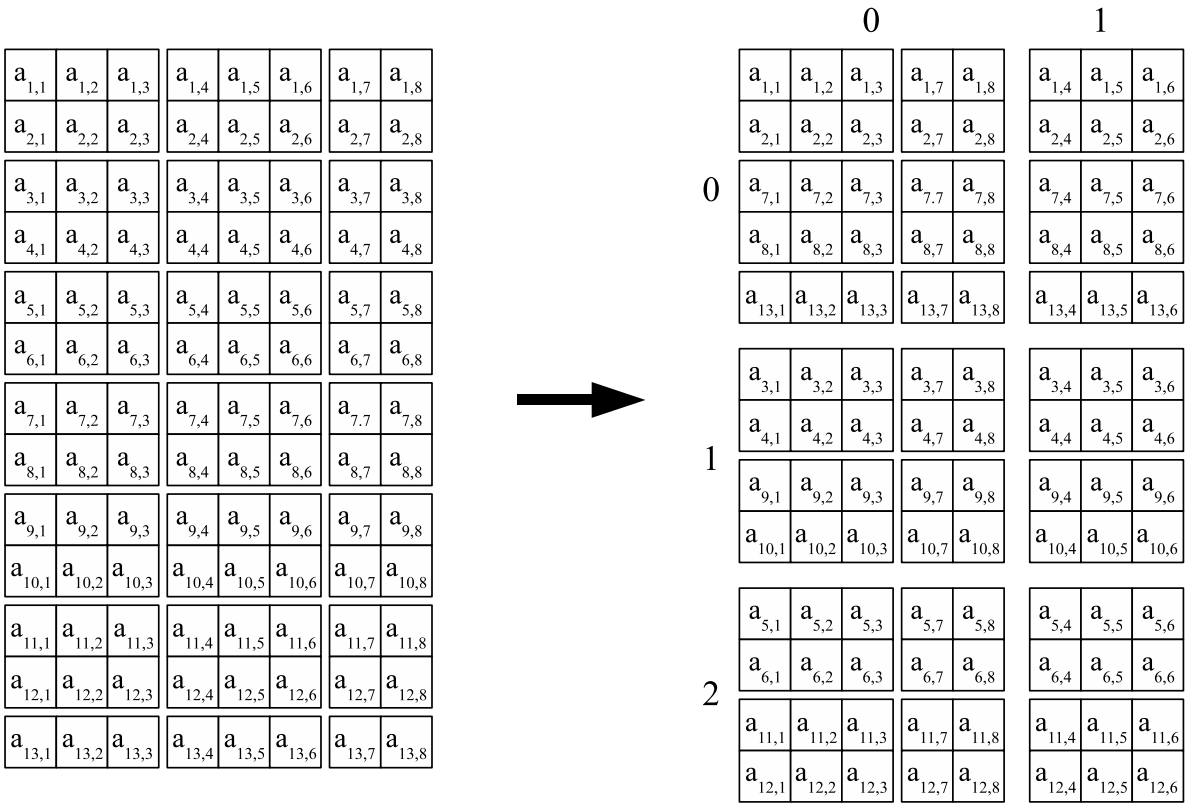

Figure 3: ScaLAPACK matrix distribution. A 13 by 8 matrix is subdivided into blocks of 2 by 3 matrix elements, and distributed onto a 3 by 2 process grid in a block cyclic manner. The numbers along the top and left edges of the process grid indicate the row and column coordinates, respectively, of each process on the grid. The blocks are separated by thin spacing, and the grid processes are separated by thick spacing.

```
for (i = 0; i < n; i++) {
  if ( (i/mb)%nprow != myrow ) continue;
  for (j = i+1; j < n; j++) {
    if ( (j/nb)%npcol != mycol ) continue;
    A[i] += f1(i, j); //Incorrect!
    N[i][i] += f3(i, j); //Incorrect!
    N[i][j] += f4(i, j);
  }
}

for (j = 0; j < n; j++) {
  if ( (j/mb)%nprow != myrow ) continue;
  for (i = 0; i < j; i++) {
    if ( (i/nb)%npcol != mycol ) continue;
    A[j] += f2(i, j); //Incorrect!
    N[j][j] += f5(i, j); //Incorrect!
    N[j][i] += f6(i, j);
  }
}
```

(7)

Restriction of the i and j loop indices in the manner depicted above parallelizes the computation because each process handles a unique subset of i and j that is selected by the `myrow` and `mycol` coordinates of that process.

Although code fragment 7 does correctly restrict the i and j addresses to only those values that correspond to the sub-vector or sub-matrix elements of a given process, this example is misleading for several reasons. First, although the first loop nest can access $\mathbf{N}[i][j]$, it cannot necessarily access $\mathbf{N}[i][i]$. Similarly, although the second loop nest can access $\mathbf{N}[j][i]$, it cannot necessarily access $\mathbf{N}[j][j]$. The diagonal elements of the matrix $\mathbf{N}$ are not guaranteed to belong to the process that calculates updates to those elements because each process owns only a sub-matrix of the distributed matrix $\mathbf{N}$, which we will call sub_$\mathbf{N}$. A solution to this problem is for each process to maintain a private copy of the diagonal elements of the matrix $\mathbf{N}$. Because $\mathbf{N}$ is a square matrix of dimensions n by n, the private copy of the diagonal elements can be stored in a vector of length n, which we will designate by the vector diag_N. When all of the processes have finished updating their private copies diag_N, all of the copies of diag_N are combined to produce the vector Q that is then rebroadcast to each process. The `MPI_Allreduce` function is perfectly suited to combining (reducing) diag_N and rebroadcasting the resulting vector Q in this manner. Upon receipt of the vector Q, each process copies from Q into the diagonal elements of its sub-matrix sub_$\mathbf{N}$ only those elements that exist in sub_$\mathbf{N}$.

The second problem with code fragment 7 arises due to a ScaLAPACK convention that requires that a distributed vector such as the vector A exist only in column zero of the

process grid. Hence, only a process that exists in column zero of the grid possesses a sub-vector of the vector A, which we will call sub_A. A particular process that calculates updates to the vector A may not lie in column zero of the grid and therefore may not be able to access the elements that it needs to update. This problem is very similar to the first problem discussed above, and it has a similar solution. Each process must maintain a private copy of the distributed vector A, which we will call priv_A. When all of the processes have finished updating their private copies priv_A, all of the copies of priv_A are combined, and the resulting vector S is rebroadcast to each process via the `MPI_Allreduce` function. Upon receipt of the vector S, each process in column zero of the process grid copies from S into the elements of its sub-vector sub_A only those elements that exist in sub_A.

The third problem with code fragment 7 is that the matrix elements $\mathbf{N}[i][j]$ and $\mathbf{N}[j][i]$, which are accessible by a given process, are not accessed as elements of the global matrix $\mathbf{N}$, but rather as elements of the sub-matrix sub_$\mathbf{N}$ that is owned by that process. Moreover, the sub-matrix sub_$\mathbf{N}$ is not addressed using the global $[i][j]$ or $[j][i]$ address directly. Instead, the global address is converted to an offset into the sub-matrix sub_$\mathbf{N}$. The address conversion is accomplished via the following `adrmap` function that requires, in addition to i and j, the number of rows or *local leading dimension* `lld` of the sub-matrix so that `adrmap` can perform column-major addressing as required by ScaLAPACK:

```
size_t adrmap( int i, int j) {
  size_t lbi, lbj;
  lbi = i/mb/nprow;
  lbj = j/nb/npcol;
  return ( lld*(nb*lbj + j%nb)
           + mb*lbi + i%mb );
}
```
(8)

Code fragment 7 may be rewritten to call the `adrmap` function as follows:

```
for (i = 0; i < n; i++) {
  if ( (i/mb)%nprow != myrow ) continue;
  for (j = i+1; j < n; j++) {
    if ( (j/nb)%npcol != mycol ) continue;
    priv_A[i] += f1(i, j);
    diag_N[i] += f3(i, j);
    sub_N[adrmap(i, j)] += f4(i, j);
  }
}

for (j = 0; j < n; j++) {
  if ( (j/mb)%nprow != myrow ) continue;
  for (i = 0; i < j; i++) {
    if ( (i/nb)%npcol != mycol ) continue;
    priv_A[j] += f2(i, j);
    diag_N[j] += f5(i, j);
    sub_N[adrmap(j, i)] += f6(i, j);
  }
}
```
(9)

After the vectors priv_A and diag_N have been updated by each process as indicated in code fragment 9, they are combined and rebroadcast to all processes by the `MPI_Allreduce` function, then copied by each process into the sub-vector sub_A and into the sub-matrix sub_$\mathbf{N}$, respectively.

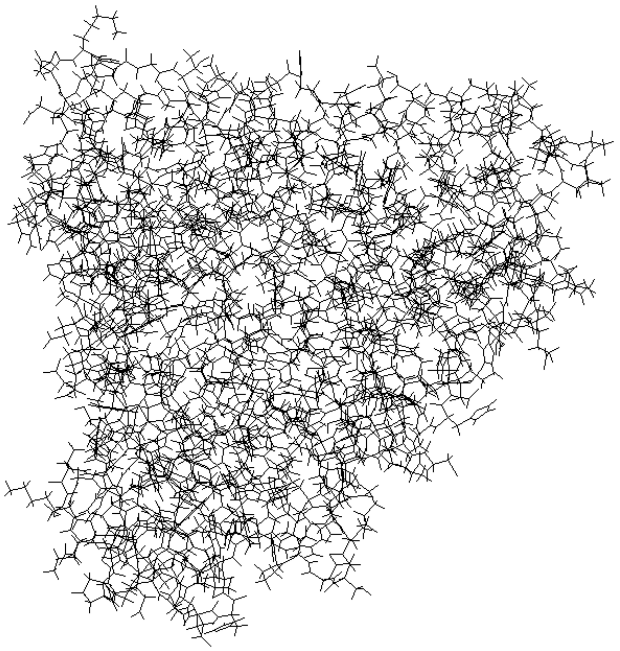

Figure 4: 6,370-atom 1AKD molecule.

Code fragments 8 and 9 demonstrate that mapping from a global matrix address to a local sub-matrix address involves several integer divisions and several modulus operations for access to *each* element of the matrix. However, because the divisors are `mb`, `nb`, `nprow` and `npcol` (which are constants for a given matrix), and further because the dividends are i and j whose values lie in the range $0 \leq i < n$, all of the divisions and modulus operations may be precomputed and stored in eight lookup tables, each of length n. Because a matrix requires $O\left(n^2\right)$ memory, these tables that require $O\left(n\right)$ memory represent only a small fraction of the size of a typical matrix, so they offer the possibility of accelerated computation at the expense of minimal additional storage.

The above discussion describes the parallelization that is accomplished via MPI for Phase 2 of the Newton-Raphson minimization. Phases 3 and 4 are executed by the `pdgemm` and `pdposv` subroutines, respectively, of the ScaLAPACK scientific library.

3. PERFORMANCE AND SCALABILITY

In this section we make scalability and performance comparisons between the OpenMP and MPI implementations of the Newton-Raphson minimization. In our performance measurements, we used the 1AKD molecule [20] from the RCSB Protein Data Bank[1]. We modified the 1AKD X-ray crystal structure by removing all of the water oxygens and other non-amino-acid atoms and then by adding hydrogen atoms, which produced a molecular model of 6,370 atoms (see Figure 4).

Figure 5 shows the scalability of Newton-Raphson minimization of the 1AKD molecule using a Sun Fire™ 15K server with 72 UltraSPARC III processors. The scalability of the MPI implementation is marginally better than the OpenMP implementation for 64 processors. This figure was

[1] `http://www.rcsb.org/pdb/`

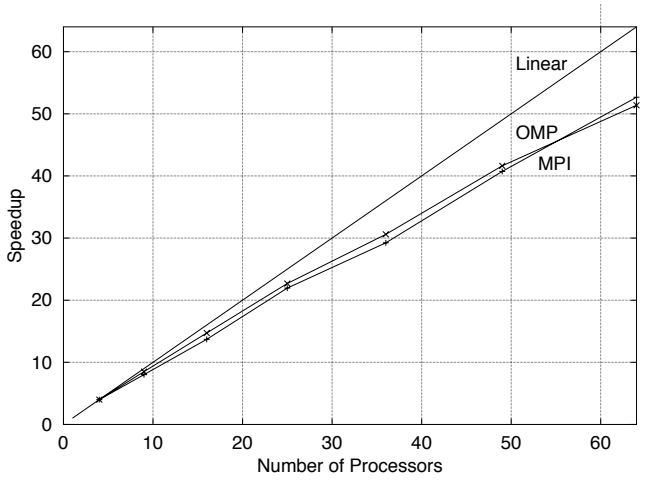

Figure 5: Scalabilities of OpenMP and MPI for Newton-Raphson minimization of the 1AKD model. The "OMP" and "MPI" plots represent the scalabilities of OpenMP and MPI, respectively. The "Linear" plot represents perfect scalability.

generated using data from Table 1. Each column of the table reports an OpenMP measurement and an MPI measurement for 4, 9, 16, 25, 36, 49 and 64 processors. (The number of processors is chosen to be an integer squared because ScaLAPACK requires a square process grid for Cholesky factorization.) The "Newton-Raphson" column shows the total execution time for two iterations of Newton-Raphson minimization. The remaining columns show the execution time for Phases 2 through 4 of Newton-Raphson minimization. The OpenMP implementation outperforms the MPI implementation by a factor of 1.3 to 2.0, as can be seen by comparing the OpenMP and MPI execution times for each row within a particular column.

4. PROGRAMMABILITY

Creating two versions of the same application using different parallelization approaches allowed us to compare the effort required to implement both versions. Parallelizing Phase 2 of the computation required the splitting of nested loops in both cases. Once this splitting was completed, the OpenMP implementation was straightforward. However, creating the MPI version required substantial additional effort (required by ScaLAPACK) to map global matrices onto a two-dimensional process grid, and to modify one-dimensional row cyclic parallelization to obtain two-dimensional block cyclic parallelization.

Whereas the OpenMP version accesses global, shared matrices and vectors, the MPI version accesses distributed matrices and vectors. Each MPI process must maintain not only a sub-matrix for each distributed matrix, but also a private copy of the diagonal elements of that matrix, as well as division and modulus lookup tables for the matrix in order to facilitate address mapping. Furthermore, each MPI process must maintain a private, complete copy of each distributed vector, as well as a sub-vector for that vector.

We have estimated the relative complexity of the two versions of NAB by counting the non-comment source code lines that are related to the Newton-Raphson minimization and to the calculation of the Born energy and its derivatives. Three categories of source code lines were counted: (1) source code lines that are required for serial execution, (2) source code lines that are required to modify the serial code for parallel execution by OpenMP, and (3) source code lines that are required to modify the serial code for parallel execution by MPI. The serial line count is 1643. The OpenMP line count is 180. The MPI line count is 962. These line counts reveal that adaptation of the serial code for MPI and ScaLAPACK produced significantly (*i.e.*, a factor of five) more source code than adaptation of the serial code for OpenMP. This finding suggests that the programmer's productivity may be higher when an application that relies on linear algebra is parallelized using OpenMP instead of MPI. Moreover, parallelization with MPI may be facilitated by parallelizing first with OpenMP. We were able to debug via OpenMP the basic code transformations necessary for parallelization prior to moving to the more challenging aspects of MPI parallelization that were required by ScaLAPACK.

5. DISCUSSION

This article is a case study of parallelizing a molecular modeling application. We have demonstrated that energy minimization computations can be implemented in a highly-scalable manner that can utilize up to 64 processors efficiently. Our experiments demonstrate that the performance and programmability of the OpenMP version are superior to those of the MPI version.

However, we recognize that the MPI version can run not only on symmetric multiprocessors, but also in distributed environments, such as Beowulf [14] clusters. Clusters may have better nominal price/performance characteristics than large symmetric multiprocessors, although cluster performance may be impeded by relatively slow interconnects typically used in those environments. A performance comparison between symmetric multiprocessors and clusters is beyond the scope of this work.

6. ACKNOWLEDGMENTS

We thank David Case, Guy Delamarter, Gabriele Jost, Daryl Madura, Eugene Loh and Ruud van der Pas for helpful comments.

The NAB software is distributed under the terms of the GNU General Public License (GPL), and can be obtained at `http://www.scripps.edu/case/` .

7. TRADEMARK LEGEND

8. REFERENCES

[1] *Sun Studio 10: Sun Performance Library User's Guide*. Sun Microsystems, Inc. http://docs.sun.com.

[2] G. M. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings*, pages 483–485, Reston, VA, 1967. AFIPS Press.

[3] D. Bashford and D. Case. Generalized born models of macromolecular solvation effects. *Ann. Rev. Phys. Chem*, 51:129, 2000.

Number of CPUs	Newton-Raphson		Phase 2		Phase 3		Phase 4	
	OpenMP	MPI	OpenMP	MPI	OpenMP	MPI	OpenMP	MPI
4	3,600.41	5,253.95	78.26	197.75	2,699.23	3,896.19	805.87	1,132.16
9	1,695.13	2,618.45	39.35	89.18	1,242.28	1,946.99	403.14	564.89
16	977.87	1,533.54	22.97	51.05	710.29	1,130.90	237.29	339.63
25	634.78	957.39	14.97	33.61	454.66	682.38	158.88	230.64
36	456.81	718.74	11.88	23.94	322.78	521.87	115.59	163.67
49	345.98	515.93	10.23	18.78	239.84	364.78	89.78	123.49
64	280.30	398.97	11.04	22.41	184.16	267.99	74.12	97.43
4/64	12.84	13.17	7.09	8.82	14.66	14.54	10.87	11.62

Table 1: Execution time (seconds) comparison between OpenMP and MPI for the 1AKD model and a Sun Fire 15K server with 72 UltraSPARC III processors. The "Newton-Raphson" column shows execution time for Newton-Raphson minimization. The remaining columns show execution time for Phases 2 through 4 of the minimization. The "4/64" row gives the ratio of the 4-processor and 64-processor execution times.

[4] U. Berkert and N. Allinger. Molecular mechanics. ACS Monograph 177, American Chemical Society, 1982.

[5] L. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongara, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R.C.Whaley. *Scalapack Users' Guide*. Society for Industrial and Applied Math, 1977.

[6] B. Brooks, R. Bruccoleri, B. Olafson, D. States, S. Swaminathan, and M. Karplus. CHARMM: A program for macromolecular energy, minimization, and dynamics calculations. *J. Comp. Chem.*, 4:87–217, 1983.

[7] R. Brown and D. Case. Second derivatives in generalized born theory. *J. Comp. Chem.*, 2006. Accepted for publication.

[8] M. Carson and J. Hermans. The molecular dynamics workshop laboratory. In J. Hermans, editor, *Molecular Dynamics and Protein Structure*, pages 165–166. University of North Carolina, Chapel Hill, 1985.

[9] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, and J. McDonald. *Parallel Programming in OpenMP*. Morgan Kaufmann, 2000.

[10] M. Feig, W. Im, and C. Brooks. Implicit solvation based on generalized born theory in different dielectric environments. *J. Chem. Phys.*, 120:903–911, 2004.

[11] R. P. Garg and I. Sharapov. *Techniques for Optimizing Applications: High Performance Computing*. Prentice Hall, PTR, 2001.

[12] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI: The Complete Reference, Vol. 2, The MPI Extensions*. MIT Press, Cambridge, MA, 1998.

[13] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface, 2nd ed*. MIT Press, Cambridge, MA, 1999.

[14] W. Gropp, E. Lusk, and T. Sterling, editors. *Beowulf Cluster Computing with Linux*. MIT Press, second edition, 2003.

[15] W. Gropp, E. Lusk, and R. Thakur. *Using MPI-2: Advanced Features of the Message Passing Interface*. MIT Press, Cambridge, MA, 1999.

[16] G. Hawkins, C. Cramer, and D. Truhlar. Parametrized models of aqueous free energies of solvation based on pairwise descreening of solute atomic charges from a dielectric medium. *J. Phys. Chem*, 100:19824–19839, 1996.

[17] T. Macke. *NAB, a language for molecular manipulation*. PhD thesis, the Scripps Research Institute, 1996.

[18] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in Fortran*. Cambridge University Press, second edition, 1992.

[19] M. Schaefer and C. Froemmel. A precise analytical method for calculating the electrostatic energy of macromolecules in aqueous solution. *J. Mol. Biol.*, 216:1045–1066, 1990.

[20] I. Schlichting, C. Jung, and H. Schulze. Crystal strucure of cytochrome p-450cam complexed with the (1s)-camphor enantiomer. *FEBS Lett.*, 415:253–257, 1997.

[21] C. Sosa, T. Hewitt, M. Lee, and D. Case. Vectorization of the generalized born model for molecular dynamics on shared-memory computers. *J. Mol. Struct. (Theochem)*, 549:193–201, 2001.

[22] J. Srinivasan, M. Trevathan, P. Beroza, and D. Case. Application of a pairwise generalized born model to proteins and nucleic acids: inclusion of salt effects. *Theor. Chem. Acc.*, 101:426–434, 1999.

[23] W. Still, A. Tempczyk, R. Hawley, and T. Hendrickson. Semianalytical treatment of solvation for molecular mechanics and dynamics. *J. Am. Chem. Soc*, 112:6127–6129, 1990.

[24] V. Tsui and D. Case. Theory and applications of the generalized born solvation model in macromolecular simulations. *Bioploymers (Nucl. Acid. Sci.)*, 56:275–291, 2001.

[25] W. van Gunsteren, H. Berendsen, J. Hermans, W. Hol, and J. Postma. Computer simulation of the dynamics of hydrated protein crystals and its comparison with x-ray data. *Proc. Natl. Acad. Sci. USA*, 80(14):4315–4319, 1983.

[26] P. Weiner and P. Kollman. AMBER: Assisted model building with energy refinement. a general program for modeling molecules and their interactions. *J. Comp. Chem.*, 2:287–303, 1981.

Implementing the CG and MG NAS parallel benchmarks in X10

[1] Vijay Saraswat
IBM Research

We discuss issues in the implementation of some NAS parallel benchmark problems (specifically CG and MG) in X10, a new language for high-productivity, high-performance computing. We show that the natural representation of these programs in X10 is quite compact. The size of the MG program, for instance, is comparable to the corresponding ZPL program, held up by some researchers as a benchmark for productivity. Additionally, in further confirmation of the results reported earlier[2], we show that the parallel and distributed versions of the X10 programs are obtained as a very small delta from the sequential version. Finally we show that these programs in fact lie within the set of those X10 programs which can statically be recognized as determinate and deadlock-free. Thus, a large class of potential concurrency-related errors can be ruled out for such programs.

[1] This work has been supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. NBCH30390004.

[2] "X10: An Object-Oriented Approach to Non-Uniform Cluster Computing", by Philippe Charles et. al., in Proceedings of OOPSLA 2005

An Experiment in Measuring the Productivity of Three Parallel Programming Languages

Kemal Ebcioğlu and Vivek Sarkar
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598, USA*

Tarek El-Ghazawi
The George Washington University
801 22nd Street NW
Washington, DC 20052, USA

John Urbanic
Pittsburgh Supercomputing Center
4400 Fifth Avenue
Pittsburgh, PA 15213, USA

Abstract

In May 2005, a 4.5 day long productivity study was performed at the Pittsburgh Supercomputing Center as part of the IBM HPCS/PERCS project, comparing the productivity of three parallel programming languages: `C+MPI`, `UPC`, *and the IBM PERCS project's* `x10` *language. 27 subjects were divided into 3 comparable groups (one per language) and all were asked to parallelize the same serial algorithm: Smith-Waterman local sequence matching – a bio-informatics kernel inspired from the Scalable Synthetic Compact Applications (SSCA) Benchmark, number 1. Two days of tutorials were given for each language, followed by two days of intense parallel programming for the main problem, and a half day of exit interviews. The study participants were mostly Science and CS students from the University of Pittsburgh, with limited or no parallel programming experience.*

There were two typical ways of solving the sequence matching problem: a wavefront algorithm, which was not scalable because of data dependencies, and yet posed programming challenges because of the frequent synchronization requirements. However, the given problem was shown to also have a subtle domain-specific property, which allowed some ostensible data dependences to be ignored in exchange for redundant computation, and boosted scalability by a great extent. This property was also given as a written hint to all the participants, encouraging them to obtain higher degrees of parallelism.

The programming activities of the study participants were recorded, both through face-to-face observations by the IBM/PSC teams, as well as through frequent automated sampling of the programs being written, such that it was later possible to analyze the progress of each study participant in great detail, including the thought process and the difficulties encountered. The detailed logs also allowed us to infer precisely when the first correct parallel solution was arrived at (this "time to first correct parallel solution" metric was used as one measure of productivity). This paper describes the results of the experiment, with particular emphasis on our technical observations on the codes produced by the anonymous participants. Interesting insights have been obtained into the problem-solving process of novice parallel programmers, including those exposing productivity pitfalls in each language, and significant differences among individuals and groups.

1 Introduction

High-performance supercomputers are heading toward increased complexity, and thus, high productivity tools and languages are very much on the agenda of supercomputing researchers. In order to be able to evaluate such productivity tools and languages with quantitative measures, there is an increased need for performing field experiments using human programmers as subjects. Feedback from such field experiments will be crucial as a guiding tool for future innovation in

*Current author contact emails: `kemal.ebcioglu@acm.org`, `vsarkar@us.ibm.com`, `tarek@gwu.edu`, `urbanic@psc.edu`.

productivity.

In this paper, we will embark on describing the technical insights into one such field experiment, comparing the parallel languages `C+MPI`[1], `UPC`[2], and `x10`[3], where the latter is IBM HPCS/PERCS project's new parallel programming language. The detailed design rationales for the languages themselves are beyond the scope of this paper, which will focus on the productivity study.

2 The productivity experiment

From Monday, May 23 to Friday, May 27, 2005, a 4.5 day long productivity study was performed at the Pittsburgh Supercomputing Center (PSC) as part of the IBM HPCS/PERCS project, comparing the productivity of three parallel programming languages mentioned above: `C+MPI`, `UPC`, and `x10`. 27 subjects were divided into 3 comparable groups (one 9-person group per language) and all were asked to parallelize the same serial algorithm: Smith-Waterman local sequence matching – a bio-informatics kernel inspired from the Scalable Synthetic Compact Applications (SSCA) Benchmarks [4], number 1. The problem was suggested to us by David Bader. The serial algorithm was provided to the subjects as a working sequential program (serial `C` for the `C+MPI` and `UPC` groups, and serial `x10` for the `x10` group).

The study participants were mostly Science and CS students from the University of Pittsburgh, mostly with limited or no parallel programming experience. The distribution of subjects to language groups were performed by the IBM Research Social Computing Group and PSC management, with no input from the technical language teams. Throughout the experiment, the subjects remained anonymous to the technical teams.

The experiment began with two days of tutorials (Monday and Tuesday) and hands-on exercises, taught by experts in each language. The availability of the PSC lab terminal and supercomputer resources were instrumental in making this possible. Then the subjects performed intense parallel programming for two days (Wednesday and Thursday) for the main problem. During the main programming session, the subjects were allowed to ask questions, but only about language constructs and technical problems they encountered, and not about the algorithmic issues themselves. All questions and the answers given were recorded. On Friday, the subjects had half a day of exit interviews summing up their experience, conducted by the IBM Research Social Computing team.

3 The problem description

The serial algorithm that the subjects were asked to parallelize is described below:

The Smith-Waterman local sequence matching algorithm consists of computing the elements of a $N+1$ by $M+1$ matrix, from two strings of lengths $N+1$ and $M+1$, where M is usually many times larger than N. Figure 1 depicts the basic computation in serial `C` code, and Figure 2 shows the resulting matrix on two example strings: `-GGTCC` and `-GCCGCATCTT`.

```
#define Match (-1)
#define MisMatch 1
#define Gap  2
int A[N+1][M+1]; //initialized to 0
char *c1=STRING1;
char *c2=STRING2;
for(int j=1;i<=M;j++)
for(int i=1;i<=N;i++)
 A[i][j]=
    MIN(0,
        A[i-1][j]+Gap,
        A[i][j-1]+Gap,
        A[i-1][j-1]+(c1[i]==c2[j]?Match:MisMatch));
```

Figure 1. The serial version of the local sequence matching algorithm

	-	G	C	C	G	C	A	T	C	T	T
-	0	0	0	0	0	0	0	0	0	0	0
G	0	-1	0	0	-1	0	0	0	0	0	0
G	0	-1	0	0	-1	0	0	0	0	0	0
T	0	0	0	0	0	0	0	-1	0	-1	-1
C	0	0	-1	-1	0	-1	0	0	-2	0	0
C	0	0	-1	-2	0	-1	0	0	-1	-1	0

C C
C C

T C
T C

Figure 2. An example computation

Among several possibilities, we will describe two typical ways of parallelizing the local sequence matching problem:

The first one is a wavefront algorithm, as illustrated in Figure 3. Since each matrix element (i, j) depends on

its West $((i, j-1))$, North $((i-1, j))$ and NorthWest $((i-1, j-1))$ neighbors, at each step, the matrix elements on a diagonal line can be computed in parallel, based on the matrix elements computed in previous steps. This algorithm and its many variants are not scalable, because of data dependencies: the maximum parallelism at any given step is limited to $min(N, M)$, the maximum length of a diagonal line (in units of matrix cells) in Figure 3; but N is small. Yet, this wavefront algorithm poses programming challenges for the novice subjects, because of the frequent communication and synchronization requirements in a distributed matrix implementation.

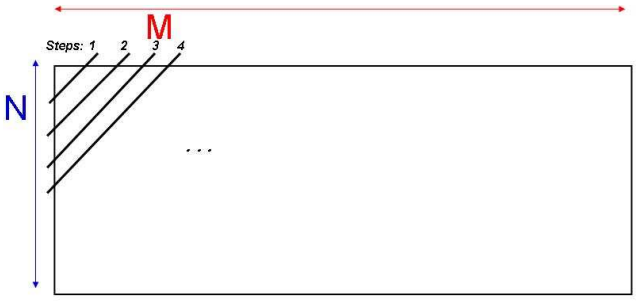

Figure 3. The wavefront computation

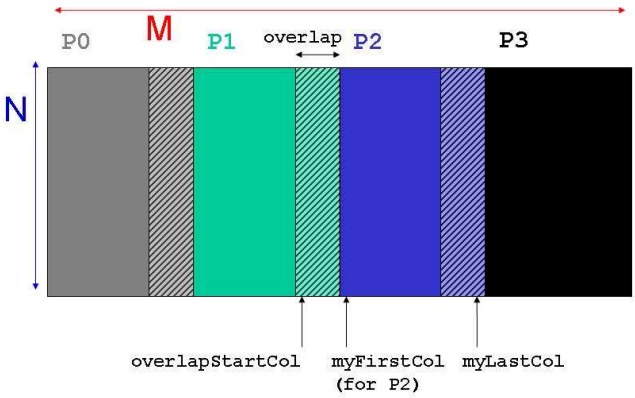

Figure 4. A scalable algorithm: Each processor computes its own columns `myFirstCol..myLastCol`, using `overlapStartCol..(myFirstCol-1)` as warm-up.

However, thanks to the bio-informatics domain experts at PSC, the given problem has been shown to also have a subtle domain-specific property, which allows some ostensible data dependences to be ignored, in exchange for redundant computation, and boosts parallel scalability by a great extent. Namely, starting the computation of figure 1 at any column of a matrix initialized to zeros (instead of column 1), and computing $N + abs((N * Match)/Gap)$ consecutive columns to the right of the start column ($1.5N$ columns in this example), ensure that the element values of the next column $(N + abs((N * Match)/Gap) + 1)$ are correct, i.e. are identical to what they would have been in the serial algorithm.

This property allows blocks of columns of the matrix to be subdivided among processors (as if with a `(*,BLOCK)` distribution), with each processor precomputing only $1.5N$ columns on the left of its block as "warm-up" (in a scratch area, without committing the results), and then, having obtained the correct element values for the leftmost column of its block, computing its entire block itself. The domain specific property mentioned above ensures that all computations are equivalent to the result of a serial computation. Figure 4 demonstrates this approach. This property was also given as a written hint to all the subjects, encouraging them to obtain higher degrees of parallelism using it.

Of course, adding wavefront computations to the scalable approach just described is also possible, for gaining additional parallelism.

4 A summary result of the productivity experiment

The productivity study team from IBM and PSC performed many automated and non-automated observations on the subjects throughout the study, by frequent sampling of the source code changes, recording of the results of compilation and execution (more than 180,000 events were automatically recorded), face-to-face observations, and interviews.

Because of the extensive automated recording of source code changes, it is possible to precisely determine when the first working parallel code was created. Figure 5 gives one summary of the study results for each subject. It shows the time between "development start time" and "development end time" as defined below, and further breaks down how that time was spent, using heuristic algorithms for identifying development phases such as authoring, debugging, and executing a program. Figure 5 also identifies which subjects never produced a working parallel program, and which subjects left the study without staying to the end.

- The *development start time* was assumed to be the first running of the serial program for the sequence

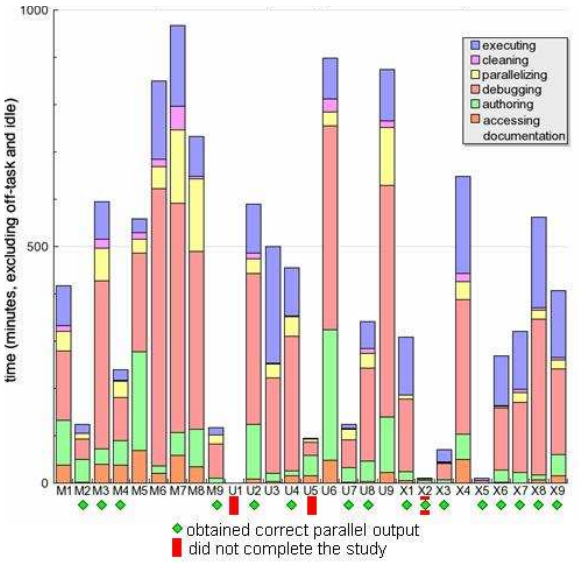

Figure 5. Development times and activities

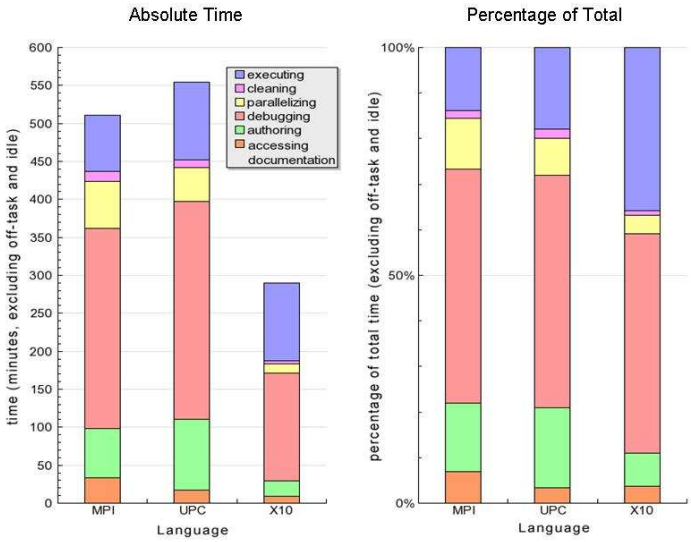

Figure 6. Development time by language

matching problem, or the appearance of the first parallel construct in the program, in case the user never ran the serial program.

- The *development end time* was taken to be the creation of a parallel program for sequence matching that gave the correct result in the PSC environment, on the reference input of $N = 10$, $M = 100$, and a fixed random pair of strings, and that indeed exhibited greater than 1X parallelism on the main computation of the matrix elements (the latter verified by a human expert). In case the subject was never able to produce such a program, the development end time was just taken to be the time the subject stopped work.

Of course, some of the subjects continued to improve and optimize their programs after creating their first working parallel program. Figure 5 does not include such optimization activities.

The additional summary result in figure 6 indicates that `x10` had an edge over `C+MPI` and `UPC` for the particular productivity metric of time-to-first-parallel-solution. Needless to say, all results are preliminary: in the conclusions section, we provide some additional insights for further study.

In the appendix of this paper, we will also focus on one additional type of analysis of the results: our technical team's own old-fashioned reading of the sequences of code produced by the subjects, which can provide some more detailed insight into the experiment. Future papers from the team will also describe other aspects of the experiment more comprehensively. The present paper is not intended to be a comprehensive presentation of the study.

5 Conclusions

Overall, we think the event that took place at PSC had the tenets of a well-designed major study. We will now summarize some of the informal observations and conclusions we have reached as the technical sub-team. We feel that these remarks may be valuable for the future experiments. While these insights are not new, it is interesting to observe their re-emergence as part of the practical productivity experiment.

Productivity insights for future language and tools design:

Abstraction mismatch: Some subjects were overwhelmed by the complexity of parallel programming (to the extent of being unable to generate any correct parallel program), or made their own work unduly complicated. The reason appears related to the inability to use the right concise abstractions, that specify what needs to be done at a high level. The parallel language and/or component library must have an arsenal of the right abstractions to express the task at hand. For example, for `x10`, it would have been better to use

an existing (*,BLOCK) distribution as an abstraction or readily available component (subject *X5* had to simulate his/her own (*,BLOCK) distribution instead). For C+MPI, it would have been better to have the abstraction to print a distributed matrix, rather than build a distributed matrix printing routine with low level synchronization and message passing. For languages using C as a base, mysterious segmentation violations (reflecting a C design trade-off that was once made in favor of performance vs. safety) are neither friendly nor can pass as a high level abstraction for a parallel programming task.

Lack of performance transparency: Tools that provide high-level, approximate feedback about the performance and parallelism being achieved by a user's program, would also be very useful to programmers at the early design stages. These tools would help recognize ostensibly parallel programs that give the correct results, but yet are not really parallel because of errors (*X4* created such a program). The tools could also provide performance awareness about language features, such as random access to arbitrary distributed array elements in UPC, or array distributions that can lead to bad locality in x10 (some x10 subjects chose a cyclic distribution for the matrix).

Lack of programming style and discipline: Programmers must not only be taught the constructs of a parallel language, but also a design style that shuns complexity and that rings mental bells of danger when a program fragment becomes too complex. Some subjects created unduly complex programs that probably reduced their potential productivity at the end. The lack of proper abstractions may have fueled this.

Lack of knowledge of parallel design idioms: Programmers must be taught the elementary parallel programming idioms such as data flow computation, divide and conquer,..., (somewhat like the design patterns of parallel programming). One UPC subject used an approach where each thread was continuously polling memory (busy-waiting), for its inputs to become available. Also, some subjects precipitated toward the comfort of having all processors do the same computation, or having all threads wait while thread 0 does all the work. Had we taught some more basic parallelism idioms up front, along with the language instruction, this could perhaps be avoided.

Nondeterminism considered harmful: Nondeterminism (available in most parallel languages, including x10) appears to be a dangerously powerful feature for novices. Nondeterminism makes it possible to write parallel programs easily, at the risk of being correct only some of the time by accident, and never noticing that in different circumstances the answer may be incorrect. The x10 team has been doing research in defining deterministic subsets of x10 to remedy this problem [5]. Also, the implementation of non-determinism must be made defensive, such that the probability of getting the right answer by accident is lowered, for example, by increasing randomness in the way threads schedulers work.

Insights for future experimental methodology and tutorials:

Tutorials are very important for getting across the basic concepts and impacting the productivity results. But sometimes it is hard for an instructor to read whether the message has gone across. The tutorials could have a non-intrusive online quiz component to them, via laptops or terminals, to provide immediate feedback to the instructors on how effective the learning has been.

Very common novice parallel programming pitfalls such as having all threads compute the same thing redundantly, or just having thread 0 do all the work, should be covered and advised against.

Correctness testing for parallel programming projects must be made rigorous, quickly testing many corner cases, and checking for deterministic stability, and should be spelled out to the subjects as well.

If at all possible, multi-day experiments must have 24-hour monitoring to accurately measure development time, not just during the day. Competitive brilliant minds will not rest, even when asked to rest during the off-hours!

Overall conclusions:

Overall, we believe that the present productivity experiment was an excellent experience. Through such a methodology, we believe that we can get one step closer to the goal of having a quantitative, measurable criterion on productivity, to guide future designs of languages and tools.

Acknowledgments

This work has been supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. NBCH30390004. First, we are very grateful to our hard-working anonymous subjects. We thank David Bader for his suggestion for using the Smith-Waterman serial algorithm in a productivity experi-

ment. We are in particular grateful to Alexander Ropelewski and Hugh B. Nicholas, the bio-informatics experts at PSC, for their help with the domain specific property that made the scalable computation of the local sequence alignment problem possible. We are also grateful to Nick Nystrom from PSC, to Christine Halverson, Catalina Danis, and Wendy Kellogg from the IBM Research Social Computing group, as well as to Mootaz Elnozahy, Ram Rajamony, and Vijay Saraswat, for making this sizable productivity experiment possible.

Appendix: Observations on codes produced by subjects

In this Appendix, we will provide our analysis of the codes produced by each of the subjects (based on old-fashioned code reading). These are preliminary, and are subject to change in the future, as we gain a better insight into the large database of programs.

5.1 X10 subjects

- *X1:* Essentially used the following algorithm: for each element (i, i) on the diagonal, $1 \leq i \leq N$, compute the element, and then in parallel do: { sequentially compute the row elements $(i, i+1 : M)$; sequentially compute the column elements $(i+1 : N, i)$ }. This is not a very parallel or scalable solution, but does exhibit a minimum amount of parallelism (less than 2X). Also, some matrix elements were computed and written more than once, which surprisingly did not give incorrect results (the multiple writes wrote the same value).
- *X2:* This subject misunderstood the hint. *X2* subdivided the matrix into blocks of $1.5N$ columns each, and computed these blocks independently and in parallel. Surprisingly (perhaps as an idiosyncrasy of the java thread scheduler at PSC), this nondeterministic parallel program gave the correct answers at PSC in a repeatable manner.
- *X3:* Another nondeterministic program (parallel loop with inter-iteration dependences) that worked correctly by coincidence.
- *X4:* This subject also misunderstood the hint. He intended to subdivide the matrix into $1.5N$ columns and process them independently. However, he/she mixed up columns and rows and created multiple threads (`x10` activities), where only one thread was computing the entire matrix sequentially, and the remaining threads were doing nothing. Also, there was a data race (nondeterminism) which could print the matrix before it was completely computed, but the data race did not actually occur at PSC. Because the parallelism was never greater than 1X, *X4*'s parallel program was disqualified.
- *X5:* Quickly created a first parallel solution using a wavefront algorithm. Remarkable sequence of programs: one can see how he/she starts with a stock wavefront pattern, encounters problems, and solves them successfully. He/she was also the only `x10` group subject who understood the hint and later used it to create a scalable and correct parallel solution. There were many if-then-else statements to optimize for different cases, which made the code unduly complex. An HPF-style `(*,BLOCK)` distribution in `x10`, which was not available at the time, could have simplified *X5*'s programming.
- *X6:* The first parallel solution was a wavefront computation.
- *X7:* Used wavefront parallelism, but with a cyclic distribution which has poor locality.
- *X8:* The program does not work when N does not divide M evenly, but works (using wavefront parallelism) when it does.
- *X9:* Uses fine-grain wavefront parallelism.

5.2 C+MPI subjects

- *M1:* No correct parallel solution. *M1*'s code generated numerous compile-time and run-time errors which *M1* never fully resolved.
- *M2:* Remarkably concise and elegant wavefront solution based on pure send-receive synchronization. Each processor is assigned a block of rows of the matrix (a `(BLOCK,*)` distribution), and sequentially performs the following for each column of its block: receive the top element of the column from the previous processor (wait if it not available yet), then compute this column from top to bottom sequentially, and then send the bottom element of this column to to the next processor. This solution is essentially a wavefront algorithm, but it is not lock-step synchronized (it is data-flow synchronized via ordinary `MPI` send and receive),

and should be able to tolerate variable latencies very well. Unfortunately, this solution will not scale, because the maximum parallelism is limited with the wavefront approach.

- *M3:* Essentially the same solution as *M2*'s.
- *M4:* Has used the hint correctly, to create a scalable parallel solution.
- *M5:* No correct parallel solution. *M5*'s initial attempt at parallelization generated a segmentation violation, which *M5* was unable to correct despite several attempts.
- *M6:* No correct parallel solution.
- *M7:* Each processor is computing the same entire matrix sequentially. This solution was not accepted as a parallel one. *M7* tried to create more elaborate parallel solutions later, but did not succeed.
- *M8:* No correct parallel solution. Generated numerous `MPI`, `malloc`, and language errors (both compile-time and run-time)
- *M9:* Has used the hint correctly to create a scalable parallel solution.

5.3 UPC subjects

- *U1:* Did not complete the study.
- *U2:* seems to have obtained perhaps the best wavefront solution across the teams, with low communication overhead. Uses a `(*,BLOCK)` distribution which eliminates vertical communication among matrix elements, and performs wavefront computation with each wave consisting of cells of size 1 by 25 (for the 10 by 100 input). But of course, the wavefront solution is not scalable.
- *U3:* Did not obtain any correct and parallel solution.
- *U4:*
- *U5:* Did not complete the study.
- *U6:* Did not obtain any correct and parallel solution. U6's initial attempt to parallelize the task was unusually complex, relying heavily on pointer arithmetic and never generating a correct solution.
- *U7:*
- *U8:*
- *U9:* Even though this UPC program is ostensibly parallel, only thread 0 is computing the entire matrix sequentially. Threads other than 0 are not doing anything. This was not accepted as a parallel solution.

References

[1] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. "MPI: The Complete Reference", Massachusetts Institute of Technology Press, 1996.

[2] Tarek El-Ghazawi, William Carlson, Thomas Sterling, Katherine Yelick. "UPC: Distributed Shared Memory Programming", Wiley Interscience, ISBN: 0-471-22048-5, 252 p., June 2005.

[3] Philippe Charles, Christopher Donawa, Kemal Ebcioglu, Christian Grothoff, Allan Kielstra, Vijay Saraswat, Vivek Sarkar, Christoph von Praun. "X10: An Object-Oriented Approach to Non-Uniform Cluster Computing", Proc. OOPSLA 2005.

[4] David Bader. "Scalable Synthetic Compact Application Benchmarks", available at `http://www.highproductivity.org/Benchmarks`. Contact the web site owners to request access to the code.

[5] Vijay Saraswat, Radha Jaghadeesan, Armando Solar-Lezama, Christoph von Praun. Determinate Imperative Programming: A clocked interpretation of imperative syntax. http://www.saraswat.org/cf.html

The SUMS Methodology for Understanding Productivity: Validation Through a Case Study Applying X10, UPC, and MPI to SSCA#1

Nick Nystrom, Deborah Weisser, and John Urbanic
Pittsburgh Supercomputing Center
{nystrom,dweisser,urbanic}@psc.edu

Abstract

Improving the designs of computers and programming environments to increase programmers' productivity requires detailed understanding of factors that hinder or promote creation of scalable software. The SUMS methodology, which applies statistical techniques to comprehensive, fine-grained instrumentation of programmers' activities, enables objective evaluation of software development processes. SUMS is explicitly designed to support high-end computing, without sacrificing generality. Through aggregation of data across multiple experiments and over time, SUMS overcomes otherwise limited sample size and broadens the scope of hypotheses that may be tested. After briefly reviewing the SUMS methodology and its current implementation, we present results from an intensive 4½-day study in which 27 subjects received training in UPC, MPI, or X10, which they then applied to DARPA's Synthetic Scalable Compact Application #1, a problem in bioinformatic sequence analysis. 180,524 raw events were recorded for the SSCA#1 task, providing a wealth of information which is analyzed statistically and graphically to test hypotheses and to identify significant relationships. This study establishes that SUMS supports robust, real-time instrumentation and that the data resulting from that instrumentation is sufficiently detailed to support meaningful statistical analyses. Having validated the SUMS methodology and its implementation, additional studies will amass the statistical weight necessary to allow empirical evaluation of programming models, environments, and practices.

1 Introduction

As high-end computational science transitions to petascale systems, demands on the scalability, complexity, and maintainability of applications will increase dramatically. Ensuring uniform load balance, hiding latency, prefetching data from remote memory, and managing code complexity will rise in importance, exceeding their already-critical roles for applications that run at only the teraflop scale. As DARPA's High Productivity Computing Systems [1] program nears its implementation phase, and as market pressures and processor and interconnect technologies converge to independently realize petascale systems, which programming models and tools will be appropriate for those systems remains an open question. Will MPI prove viable to manage hundreds of thousands of processes, or will explicitly parallel languages be necessary to manage parallelism and complex memory systems? If the latter, what costs can project managers expect for developing new applications and for rewriting legacy applications, and what will be the performance gain? What are the characteristics of tools necessary for debugging and optimizing performance at those scales? As a community, we have significant experience with message passing and shared memory paradigms at smaller scales, but we have very little experience with partitioned global address space (PGAS) languages, e.g. UPC [2], Co-Array Fortran [3], and Titanium [4], or even newer, experimental languages such as X10 [4], Chapel [6], and Fortress [7], and we have essentially no experience managing $O(10^5–10^6)$ threads. While experience gained on today's systems is undoubtedly valuable, it does not provide an adequate foundation from which we might soundly extrapolate software engineering techniques to systems that are larger by two orders of magnitude.

SUMS [8] is a methodology for understanding factors affecting programmers' productivity, together with an implementation of that methodology. SUMS instruments programmers' activities comprehensively and at high resolution, producing minutely detailed data ideally suited for both real-time and offline analysis. Instrumentation is unobtrusive, so as to avoid perturbing programmers' natural workflows. The detailed data support testing of hypotheses and generation of inferences through statistical analyses, and as the volume of data grows, techniques from data mining and statistical learning will become applicable. Through SUMS, we seek to learn which elements of architecture, programming models and languages, tools, systems software, and hardware design contribute to system productivity, and how. Data may be gathered from

classroom/workshop settings and ongoing research development, reflecting designed experiments as well as actual software development. Analyses may then select subsets of data, aggregating data to span different programming tasks, architectures, models, languages, hardware, and software. This statistical treatment of objectively gathered programming data produces quantitative inferences with statistically meaningful interpretations and bounds.

2 Architecture and Implementation

The SUMS architecture features distinct layers (Figure 1): acquisition components, deployed at productivity experiments to acquire raw data; analysis and discovery components, which implement techniques of data mining and machine learning to cluster, recognize patterns in, and draw inferences from the SUMS data; and presentation components, which provide a user interface through which analysts explore and interact with the data. Complementing the architecture, the SUMS database efficiently and securely bridges data acquisition and analysis layers. Implementation of the data acquisition components has matured slightly from that presented in [8]; however, the overall architecture and design goals of SUMS are unchanged.

2.1 Summary of Studies Supported by SUMS

Table 1 summarizes studies in which SUMS was used to instrument programmers' activity. Each validated a successive stage in the development and deployment of the tools which ultimately must run reliably and with minimal impact on their host systems in production environments. These studies were conducted on TCS, a 3000-processor AlphaServer SC system (Tru64 OS, 2-rail Quadrics), which was simultaneously running in full production at the Pittsburgh Supercomputing Center. Specific queues were configured to provide study participants with reasonable turnaround times.

3 SUMS Experiment 3: Contrasting MPI, UPC, and X10

In this paper, we focus on the third study in which SUMS was deployed, which was held at the Pittsburgh Supercomputing Center on May 23-27, 2005. The goal of this study was to contrast programmer productivity in MPI, UPC, and

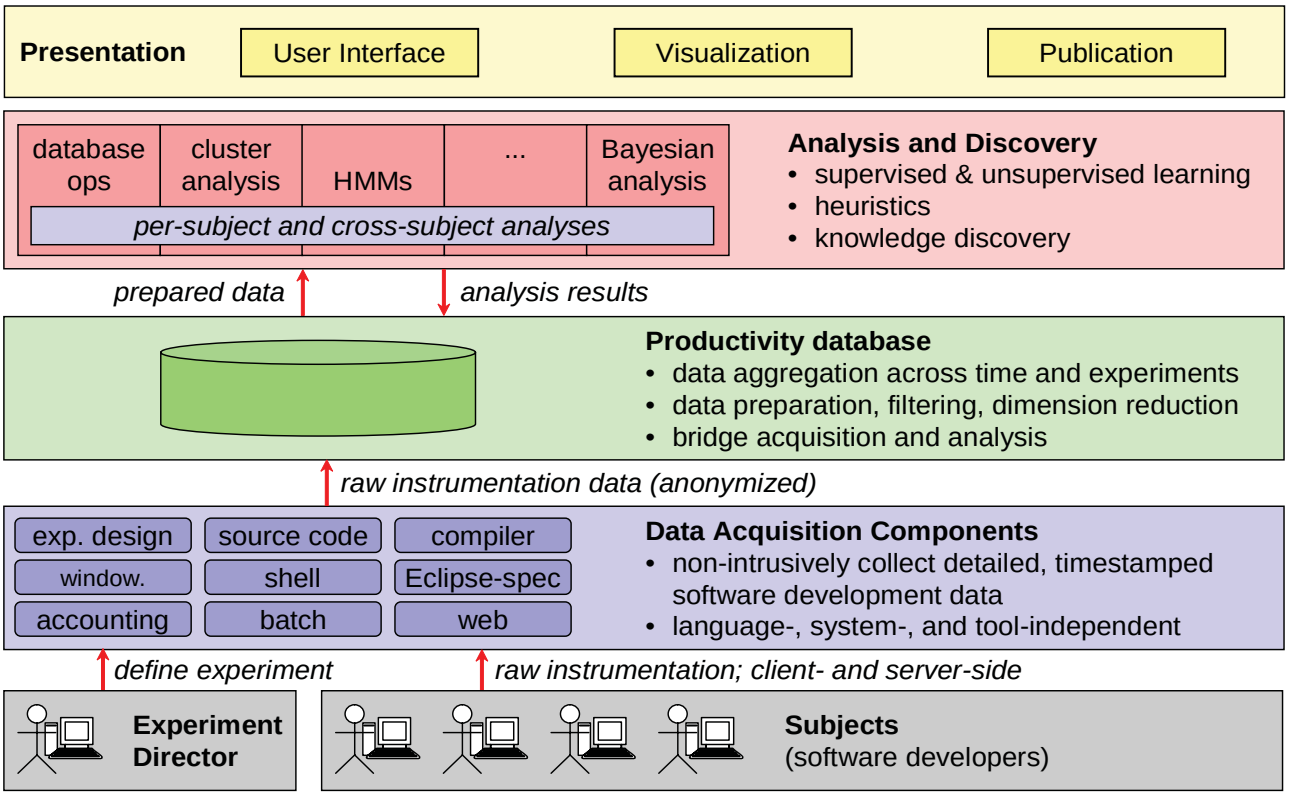

Figure 1. SUMS architecture.

Table 1. Summary of productivity experiments 1, 2, and 3.

experiment	subjects	programming model(s)	programming exercise	milestones
1. Workshop: *Introduction to Terascale Code Development* (PSC, 9/13-14/04)	4 undergraduates 7 graduate students 2 postdocs 2 faculty 4 research staff 19 total	MPI: C, Fortran	• Solution of Laplace's equation • assigned from serial reference codes • C: 72 lines of code; Fortran: 82 lines of code	• SUMS proof-of-concept • Successful deployment of data acquisition components • Successful instrumentation of development activity: 19,717 actions recorded
2. Class: *Parallel Programming* (Pitt, 11/29-12/2/04)	10 undergraduates	MPI, OpenMP: C	• Iterative Jacobi solution of a tridiagonal system • assigned as pseudocode • serial reference implementation: 45 lines of code	• Successful deployment at an external institution • Designed experiment: 5 subjects performed the exercise with OpenMP while the other 5 used MPI. The groups then switched to the other programming model, potentially allowing for discrimination of learning effects
3. Study: *Implementing the Smith-Waterman Algorithm SSCA#1 Using X10, UPC, and MPI* (PSC, 5/23-27/05)	27 undergrad and graduate students	X10, UPC, MPI	• parallelize the Smith-Waterman algorithm described in SSCA#1 • a serial solution was provided	• Large, homogeneous subject pool • 4.5-day duration, controlled environment • Framework and methodology for unobtrusively and exhaustively instrumenting development and quantitatively analyzing resulting data • External deployment and study of X10 • Programming task represents a class of algorithms of interest to DARPA mission partners

X10 for a particular programming task, the parallelization of the alignment portion of the Smith-Waterman algorithm (SSCA#1). This programming task was selected because it can be described to a general audience in a succinct lecture, and it represents a class of algorithms of interest to DARPA mission partners. Analysis of SUMS data gathered during the experiment included the inference of development timelines for each subject, which in turn yielded distinctions between different programming languages. Earlier experiments are discussed in detail in [8]. Additional productivity studies are planned, addressing a range of programming models and subject backgrounds.

In experiment 3, 27 subjects participated in a comprehensive study consisting of 48.5 hours on-site over 4.5 days at PSC. The subjects were screened for experience in C and/or Java and divided evenly by experience into 3 groups of 9, one group for each of MPI, UPC, and X10. Environmental factors were minimized as much as possible: All subjects were given the same programming task. Programming environments were the same, excepting compilers. Expert training and support were provided for each language. Meals were provided on-site, and instruction rooms were selected to be equidistant from the training center. Data were collected in two ways: through instrumentation via SUMS, and through observations and interviews by Christine Halverson, Catalina Danis, and Justin Weisz of IBM's Social Computing Group [9].

The first two days of the experiment included lectures and exercises to establish baseline familiarity with the programming environment and task. The subjects first attended a lecture on parallel programming followed by three tracks of language-specific instruction. The subjects were given coding exercises and then attended a lecture on the Smith-Waterman algorithm. On the third, fourth, and fifth days, the subjects worked on the programming task, parallelizing a serial implementation of the Smith-Waterman algorithm. Additional exercises were provided for those who finished early. The experiment ended with debriefings and interviews.

Initial preprocessing and anonymization of raw data yielded 180,524 time-stamped, categorized events, which were stored in the SUMS database. Each event contains detailed information which was then used for contextual and temporal analysis. The data relevant to this study includes:

- *Source code*: full source; differences between versions; classification of individual changes into tasks (parallelizing, serializing, debugging, commenting, cleanup, unassigned); SLOC (using sclc); subsequent classification into parallel vs. serial, runs correctly vs.

incorrectly; subsequent analysis of correctness and measurement of execution time
- *Compiler*: all compiler invocations; command lines; compiler output; errors and warnings; number of processors (when specified)
- *Batch system*: submission, execution, and completion times; number of nodes and processors; job name; queue; queue delay; system-specific resources
- *Web access*: URL; classification into task (accessing documentation, off-task)
- *Window focus*: application name; application class; classification into window type (terminal, browser, debugger, IDE, performance, system)
- *Process accounting*: commands, processors, execution time
- *Runtime output*: subjects' X10 output obtained from compiler logs; MPI and UPC output generated for each source version using the assigned 10x100 problem
- *Shell*: commands

The data collected is summarized in Table 2.

Through subsequent analysis of SUMS data, including program output and source code [10], development time was segmented into up to 5 milestones for each subject: the time at which the subject copied the provided serial code, the time the first parallel construct was inserted into the source code, the time the first correct serial output was obtained, the time the first correct parallel output was obtained on four processors, and the time of the final recorded source, compile, or batch event, or the time at which the subject left the study. For this study, "correct output" was defined to require correct results for the 10×100 alignment and normal termination of the program. Total development time was defined as beginning the first time the subject obtains correct serial output or copies the serial code and as ending the first time the subject obtains correct parallel output, or when the subject stops developing code.

By analyzing the time-stamped events for each subject in the SUMS database, e.g. source code changes, program output, and compiler output, 5-minute intervals were categorized into combinations of 8 types:

- *Accessing Documentation*: Access to web pages pertaining to languages of interest, parallel programming, or other educational content; access to man pages or online tutorials
- *Debugging*: Code modifications indicative of debugging: code changes, compilation, execution following an unsuccessful compile or run.

Table 2. Instrumented raw events for the full SSCA#1 experiment.

user_id	source	src_diff	compiler	batch	shell	web	window	total
M1	132	353	179	168	1,305	9,775	2,450	14,362
M2	155	366	151	463	1,975	346	2,259	5,715
M3	237	465	330	107	1,386	2,736	2,483	7,744
M4	69	134	81	30	432	3,801	1,664	6,211
M5	119	271	139	24	608	299	1,458	2,918
M6	327	313	287	79	1,235	366	2,658	5,265
M7	409	1,067	427	77	1,920	1,300	3,691	8,891
M8	258	766	342	73	1,355	3,250	2,504	8,548
M9	116	247	160	59	875	913	1,224	3,594
U1	129	145	254	138	1,485	20	416	2,587
U2	224	525	256	240	1,669	733	2,222	5,869
U3	236	449	268	427	8,053	1,014	4,204	14,651
U4	316	420	162	298	1,301	580	1,478	4,555
U5	82	63	20	24	274	486	653	1,602
U6	297	388	179	227	1,479	389	1,691	4,650
U7	207	661	419	104	1,625	7,396	1,550	11,962
U8	244	500	238	303	7,529	645	1,563	11,022
U9	422	847	402	342	2,492	1,793	2,268	8,566
X1	767	354	645	0	2,104	987	2,056	6,913
X2	162	228	404	0	1,109	30	687	2,620
X3	766	329	432	0	1,421	91	1,419	4,458
X4	236	420	455	0	1,341	1,007	3,518	6,977
X5	680	251	669	0	1,809	844	1,753	6,006
X6	291	348	663	0	1,809	1,446	1,661	6,218
X7	238	484	595	0	1,731	307	1,396	4,751
X8	405	452	582	0	1,727	888	2,465	6,519
X9	217	319	887	0	2,634	1,025	2,268	7,350
total	7,741	11,165	9,626	3,183	52,683	42,467	53,659	180,524

- *Parallelizing*: Adding parallel constructs to source code or repeated execution while scaling data or processors
- *Authoring*: Adding code when not in "Debugging" or "Parallelizing" states
- *Cleaning*: Commenting, reformatting
- Executing: Running the application, either as a batch job or interactively
- *Off-task*: Time unrelated to the assigned task: reading email, visiting unrelated URLs
- Idle: Time during which no activity was instrumented

Development time is defined to include only on-task activities: Accessing Documentation, Debugging, Parallelizing, Authoring, Cleaning, and Executing. Development timelines for each subject are shown in Figure 2.

3.1 Analysis

In Figure 3, we see that in comparing average development times between languages, the average total development time for subjects using X10 was significantly lower than that for subjects using MPI or UPC. The relative time spent debugging was roughly the same across all three languages. Subjects using MPI spent more time accessing documentation (tutorials were online; more documentation is available), whereas X10 programmers spent relatively more time executing code and relatively less time authoring and debugging code. Also, because at the time of this study X10 was a research language translated to Java, X10 programs were executed interactively, whereas MPI and UPC programs were submitted to a batch system. This difference in execution environments introduced an unavoidable but significant difference in subjects' workflows.

Figure 4 shows the larger maximum and median times to correct parallel outputs as well as the increased variability in development times. Of the subjects who obtained correct parallel output, those using MPI succeeded in 117 to 594 minutes with median 182 minutes. Subjects using UPC showed similar minimum and maximum times, 125 minutes and 590 minutes, respectively, but their median time was approximately twice as high, 399 minutes. Subjects who obtained correct parallel output using X10 did so in 10 to 562 minutes, with median 289 minutes. Examination of source code snapshots [10] revealed that the remarkably rapid 10 minute solution with X10 was in fact nondeterministic and would not extend to general se-

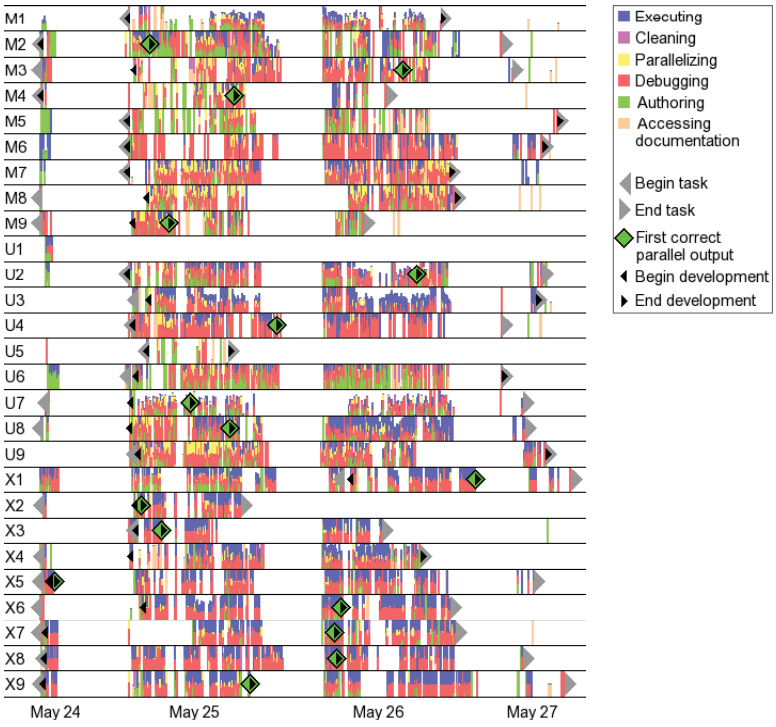

Figure 2. Development timeline for each subject. Each vertical bar depicts 5 minutes of development time, colored by the distribution of activities within the interval.

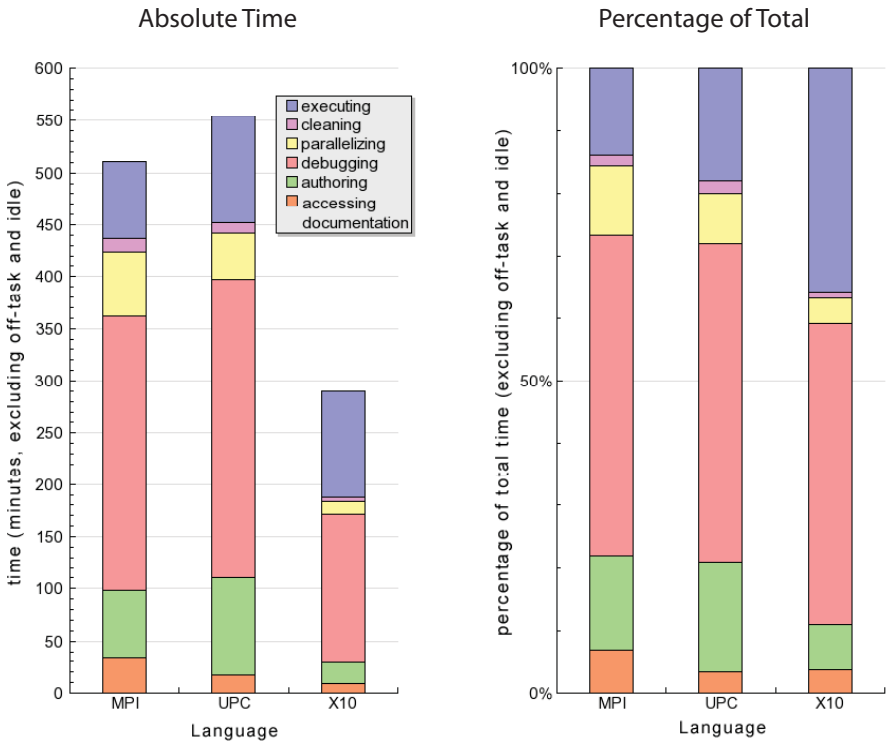

Figure : Average development time by language.

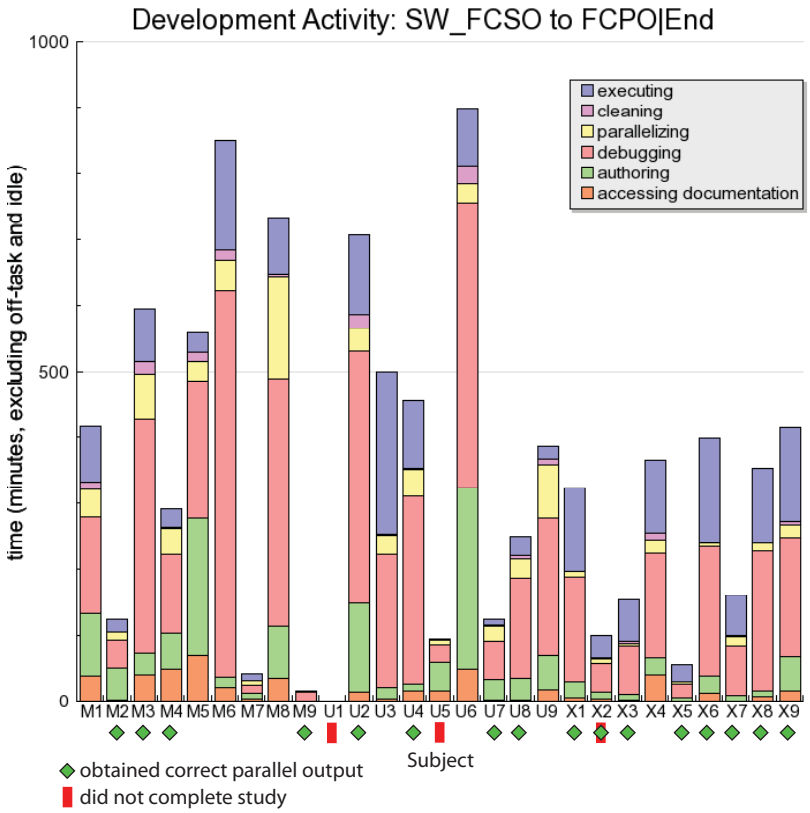

Figure 3: Development time from serial output (provided) to first correct parallel output (where obtained) or to end of study (where no correct parallel output was obtained).

quence alignments; however, it did produce correct results for the specified input.

Interestingly, of the 9 subjects in each group, the distribution of those producing correct parallel output within the alotted time was nonuniform. 4 succeeded with MPI, 4 with UPC, and 8 with X10. Whether this observation reflects properties of the languages or whether it is due to differences in programming environments or individuals' talents would require expansion of the sample size.

Collaborating on this study, C. Danis and C. Halverson applied observational results to provide insights into periods during which no activity is instrumented ("gap analysis"), and observational data were also integrated with data automatically collected by SUMS. These techniques, which can lead to insights regarding programming habits and difficult-to-instrument actions (e.g. visual focus among multiple windows), are discussed extensively in [9].

Also collaborating, V. Sarkar, K. Ebcioglu, and T. El-Ghazawi examined every source code snapshot to identify the algorithms the subjects implemented and to analyze their correctness and parallelization technique. This analysis led to insights regarding nondeterminism that would guide the evolution of X10, and it revealed subtleties of the source codes that would be difficult to ascertain automatically. V. Sarkar and K. Ebcioglu present their analyses in [10].

The three approaches of quantitative and objective instrumentation, direct observation, and source code analysis are complementary. However, there is a fundamental difference in their scalability because both observation and source code analysis are extremely labor-intensive. Achieving statistically significant results that overcome differences in individuals' backgrounds and skill will require large sample sizes, and broadening inquiry to include realistic research applications mandates studies that span months to years, not days. Automated instrumentation scales to arbitrary numbers of experiments over any duration; however, manual observation and manual source code analysis will have to be coupled selectively, where specific insights are desired.

4 SUMS 1.0

SUMS 1.0 supports the following data acquisition components, supporting both Linux (server- and client-side) and Windows (client-side).

- **Source Logger**
  Implementation: Script invoked by source daemon to capture timed, incremental snapshots and by the compiler component to capture snapshots before every build.
  Strength: Frequent snapshots of all relevant files assure capture of complete code development history.

- **Compiler Logger**
  Implementation: Wrappers in the SUMS bin directory, configured into the user's path via shell startup files.
  Strength: Capturing complete compiler options and output facilitates determination of code state; e.g. fundamental syntax errors vs. subtle warnings.

- **Shell Command Logger**
  Implementation: An executable pre-loader intercepts all executables, recording commands and arguments.
  Strength: Instruments the most basic level of developer activity; man page access; tracks use of arbitrary tools; redundancy with other components (corroboration)

- **Window Focus Tracker**
  Implementation: Lightweight process monitors window manager events; X Windows and MS Windows.
  Strength: Relate attention to active window; follow usage patterns within debuggers and performance tools; identify client-side activity (editors, local compilation, …) and off-task activity (e-mail, chat, non-development applications).

- **Web Access Logger**
  Implementation: Squid proxy.
  Strength: Track accesses to documentation, course materials, and other on- and off-task web documents.

- **Batch System Logger**
  Implementation: Built-in PBS job recording utilities extract job timing and size information.
  Strength: Record execution scripts, environment (number of processors, environment variables, …), and runtime, enabling analysis of performance and scalability.

5 Conclusion and Future Work

We make use of SUMS to understand programmer productivity by transparently obtaining fine-grained, comprehensive data spanning the software implementation process. Three initial productivity experiments provided seed data, which we are now using to establish effective data mining techniques. Additional experiments will increase SUMS data to statistical significance. We are actively seeking new participants in those and in research settings to improve sampling. Experimental systems, programming models and languages, and problem domains will be expanded as new architectures, compilers, and development environments become available.

Due to the large volume of high-dimensional data,

visual tools are helpful for initially identifying features and trends, after which carefully constructed queries yield the most fruitful analyses. (As productive lines of inquiry are identified, automated analyses are being developed.)

Possible areas of inquiry for future experiments include exploration of different tools and environments, e.g. IDEs; parallelization approaches, e.g. parallelization of serial code vs. ab initio parallelization and different strategies for data distribution; different languages, models, architectures, exploration of individual language constructs; instructional techniques conducive to generating comparable results, e.g. goal-oriented vs. broad language overviews.

5 Acknowledgments

SUMS is supported by IBM through PERCS [Grant number W01229580], in collaboration with Rami Melhem and Raymond Hoare (University of Pittsburgh). We also thank Vivek Sarkar, Kemal Ebcioglu, Vijay Saraswat, and Tarek El-Ghazawi for their contributions in the experiment, which included the X10 and UPC compilers along with teaching the X10 and UPC. We would also like to thank Ram Rajamony, and Mootaz Elnozahy (IBM Research) for insightful discussions, Alex Ropelewski for presenting the Smith-Waterman algorithm, and Victor Puchkarev and Courtney Machi for their contributions to SUMS software and analysis.

6 References

[1] DARPA High Productivity Computing Systems program. http://www.darpa.mil/ipto/programs/hpcs/

[2] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren, Introduction to UPC and language specification, Technical Report CCS-TR-99-157, Center for Computing Sciences, Bowie, MD, May 1999.

[3] R. W. Numrich and J. K. Reid, Co-Array Fortran for parallel programming, Technical Report RAL-TR-1998-060, Rutherford Appleton Laboratory, Oxon, UK, August 1998.

[4] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken, Titanium: A high-performance Java dialect, in ACM 1998 Workshop on Java for High-Performance Network Computing, 1998.

[5] The X10 Programming Language, http://domino.research.ibm.com/comm/research_projects.nsf/pages/x10.index.html

[6] Chapel: The Cascade High-Productivity Language, http://chapel.cs.washington.edu/

[7] E. Allen, D. Chase, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele Jr., and S. Tobin-Hochstadt, The Fortress Language Specification, v0.785, http://research.sun.com/projects/plrg/fortress0785.pdf

[8] Understanding Productivity through Non-intrusive Instrumentation and Statistical Learning. In The Second Workshop on Productivity and Performance in High-End Computing (P-PHEC-2), San Francisco, CA, 2005. http://www.research.ibm.com/arl/pphec/P-PHEC-2005.html

[9] C. Danis and C. Halverson, The Value Derived from the Observational Component in an Integrated Methodology for the Study of HPC Programmer Productivity. In The Third Workshop on Productivity and Performance in High-End Computing (P-PHEC-3), Austin, Texas, 2006.

[10] V. Sarkar and K. Ebcioglu, An Experiment in Measuring the Productivity of Three Parallel Programming Languages. In The Third Workshop on Productivity and Performance in High-End Computing (P-PHEC-3), Austin, Texas, 2006.