The Last Mile: Parallel Programming and Usability

Caitlin Sadowski
Computer Science Department
University of California at Santa Cruz
Santa Cruz, CA, USA
supertri@cs.ucsc.edu

Andrew Shewmaker
Computer Science Department
University of California at Santa Cruz
Santa Cruz, CA, USA
shewa@cs.ucsc.edu

ABSTRACT

Multiprocessors are now commonplace, and cloud computing is swiftly following suit. While it is possible to write high performance code for these systems, concurrency bugs are extremely common and theoretical performance is often difficult to realize. In order to take advantage of increasing numbers of parallel resources, numerous parallel programming systems have been proposed and deployed, usually without a systematic evaluation of their usability. In order to make both programmers and their parallel applications more effective, we need more useful metrics for measuring programmer productivity and a better way to evaluate such metrics. We posit that usability is a key factor in the effectiveness of a parallel programming system, and that theoretical performance gains can only be realized if programmers are able to successfully reason about their parallel code.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*parallel programming; distributed programming*; H.1.2 [**Models and Principles**]: User/Machine Systems—*human factors; software psychology*

General Terms

Human Factors, Languages, Measurement, Performance

1. INTRODUCTION

Until recently, processor speeds would double every 18 months alongside transistor densities. Now the industry has hit a power wall that makes increasing the processor speed untenable [ABD+09]. Since processor clock speeds have plateaued, exploiting multiple levels of parallelism is now the primary way to enhance performance; parallel processors are now found in home computers as well as big data centers and national laboratories. Although parallel programming has spread, getting good performance while coordinating tasks with locks, messages, and other techniques continues to be devilishly hard. Competing paradigms, languages, teaching styles, and tools have been debated in the research literature.

Usability is a measure of how easy a system is to use in terms of learnability, efficiency in the hands of an expert, ease of making mistakes, and user satisfaction [Nie]. There are a plethora of parallel programming systems (including languages, libraries, IDEs and correctness checkers); many of these have neglected to focus on usability. We posit that usability is a key factor in the effectiveness of a parallel programming system, and that theoretical performance gains can only be realized if programmers are able to successfully reason about their parallel code.

1.1 Contributions

In this position paper we outline the past and future of research on parallel programming on usability:

- We present an overview of related work on parallel programming languages and usability (Section 2).
- We outline a selection of research challenges for measuring productivity in a parallel programming language (Section 3). We provide a selection of metrics for programmer productivity (Section 3) to serve as fodder for future discussions on this topic.
- We identify five further research directions that may help to improve the usability of parallel programming systems (Section 4).

2. RELATED WORK

Although the term "Parallel Programming System" encompasses IDEs and other software engineering contexts, most prior research on productivity and parallel programming systems has focused on languages or libraries for writing parallel programs. For this reason, we consider that research on productivity and parallel programming systems lies in the intersection of programming languages (PL) and human computer interaction (HCI).

Research that falls at the intersection of PL and HCI is starting to become something of a hot topic [MK09], as increasing numbers of researchers are looking to HCI methods to help improve software engineering practices. As just one example, it seems likely that the utility of parallel debuggers could be improved by applying HCI data mining techniques to identify and add support for common debugging strategies [FKGB10, GBR10]. Greg Morrisett lists closing the gap between HCI and PL as one of the grand challenges in PL research [gra09].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
FoSER 2010, November 7–8, 2010, Santa Fe, New Mexico, USA.
Copyright 2010 ACM 978-1-4503-0427-6/10/11 ...$10.00.

A programming language is, by definition, a form of interaction between humans and computers. The implicit goal of PL research is to make it easier for people to write correct, scalable, understandable code. With this research goal, one might imagine the two subfields of HCI and PL share more past research than they do. However, little work has been done at the intersection of parallel programming and usability, and the small body of research that does exist in this area is mostly inconclusive [Luf09, HC09, SSE96, ESEG+06].

One study [RHW10] with a notably large number of participants involved 237 undergraduate students implementing the same program with coarse-grain locks, fine-grain locks, monitors, and transactions. They found that students consistently thought locking was easiest to think about and had the best syntax, but their subjective impressions of transactional memory improved significantly with experience (and were on par with fine-grained locking by the second year of exposure to transactional memory). Regardless of familiarity, the transactional memory implementations had the fewest errors. In other words, subjective impressions of the difficulty of a particular implementation strategy are not necessarily correlated with correctness.

Szafron and Schaeffer [SSE96] compared a message passing library to a GUI-based parallel programming system called Enterprise. The participants for this experiment were graduate students completing an assignment. In this study, the messaging passing library implementations were faster and were coded in less time, but the Enterprise implementations required fewer lines of code (LOC) and fewer compilations. The researchers note that each model encourages the programmers to make different kinds of mistakes, although the Enterprise mistakes are rated as less serious (tending to decrease efficiency but not correctness). This study highlights the difficulty in reconciling comparisons made along different axes.

Ebcioglu *et al.* [ESEG+06] compared three different languages, and found that the X10 language had a shorter development time, defined as time to either a correct solution or giving up on the problem; the authors do not report if this result is statistically significant. What is particularly interesting about this study is that about a third of the participants in all three groups did not successfully complete a correct solution that exhibited *any* speedup. Several of the remaining participants' solutions had scalability issues. This study demonstrates that parallel programming is difficult in all three of the languages under consideration.

2.1 Usability and HPC

Although most programmers are employed outside of the large scientific computing centers, High Performance Computing (HPC) systems act as a forecast for future business and home systems; the scale of parallelism HPC systems achieved on large, expensive machines in past generations is now accessible to a much wider base of programmers. HPC programmers also make up an important community of parallel programmers.

The DARPA High Productivity Computing Systems program [hpcb, HC09] seeks to improve the state of scientific computing. Several new languages are currently in development, and the initiative has sparked renewed interest in usability issues. As a result of this initiative, a growing group of researchers have been looking at programmer productivity for developing HPC software [HC09]. A collaborative of universities are conducting a variety of classroom studies to compare different programming models across a variety of representative applications [HCS+05]. It is worth noting that most of these studies use novices as the study participants rather than experienced HPC programmers.

Meredydd Luff [Luf09] compared multithreading with the actor model (*i.e.* message passing), transactional memory, and a sequential control sample. Each participant wrote code for the multithreaded case and one of the other cases. This work measured programmer effort, using the three main metrics of time taken to write the program, LOC, and subjective impressions. The end result was inconclusive.The learning effect was significant; the second implementation of each participant was significantly faster to code. Also, participants fell into two distinct groups of either finishing their code in time, or not finishing at all. This study highlights the fact that there is substantial variation between programmers.

Hochstein *et al.* [HCS+05] compared serial, MPI (message passing), and OpenMP (shared memory) implementations, written by novice programmers, of two problems. They found that MPI effort (in terms of time and LOC) was greater than OpenMP effort, but there were not enough data points to compare the program performance in a statistically meaningful way. They also measured the time taken to transform a serial implementation into a parallel implementation. Hochstein *et al.* [HBVG08] later compare the parallel random access memory (PRAM) model, which supports synchronous parallel operations to avoid common concurrency errors like data races, with MPI message passing. They found that the PRAM-like implementations were faster, but there was not a statistically significant difference in program correctness. These studies demonstrates the difficulty of designing experiments comparing programmer effort which result in a statistically significant comparison.

The HPC Challenge Award Competition [HPCa] recognizes the most productive implementations of a set of benchmarks, weighted 50% on performance and 50% on subjective impressions of code elegance, clarity, and size. This competition highlights the often dramatic tradeoff between code elegance and code performance. For example the LOC used to implement the Linpack benchmark in C/MPI[1], Cilk[2], and Chapel[1] were 8800, 348, and 176, respectively. Reports on the Cilk and Chapel implementations [Kus06, CCZ07] go into detail about the language features that reduce the amount of code needed. Cilk (or its Altix platform) had difficulty scaling up the performance of Linpack.

Exascale systems are expected in the 2018-2020 time frame, bringing additional requirements for resilience and power awareness in HPC code. Although programs like High Productivity Computing Systems are positive developments, considerably more resources continue to be devoted to buying new hardware instead of improving the software to effectively take advantage of existing hardware. The usability of HPC parallel programming systems is becoming even more challenging than it already is.

2.2 Usability and Parallel Abstractions

A few studies have started to look at usability issues and core parallel programming concepts. Eccles *et al.* [ES05] use a card sorting technique to get both novices and ex-

[1]From the 2009 competition.
[2]From the 2006 competition.

perts to categorize different parallel algorithms. They found that novices and experts each used a different classification scheme. They conclude that making the core features of the expert classification scheme (*i.e.* communication granularity and associated overheads) more obvious within a parallel programming system may increase the usability of that system. This difference in classification could identify a set of concepts which delineate novices from experts. McKenney *et al.* [MGM+09] propose a framework for describing the inherent difficulty in parallel programming in which parallel programming challenges are divided into four areas: work partitioning, parallel access control, resource partitioning, and interacting with hardware.

A large portion of parallel programming uses the multithreaded shared memory concurrency model, which has been heavily criticized for being difficult to reason about. Traditional threads and locks, though popular, are "incomprehensible to humans" [Lee06]. Some new programming frameworks (*e.g.* OpenCL, .NET 4.0) provide abstractions built on top of thread pools. These abstractions are less general but more easily understood. Abstractions also increase productivity (and possibly code correctness) at the expense of performance. Unfortunately, the exact magnitudes of this tradeoff are often unclear.

2.3 Synthesis

Although most of the research focused on parallel programming and usability is inconclusive, it is still possible to draw some conclusions. The research demonstrates that parallel programming is difficult, and that writing correct parallel programs is even more difficult. In fact, writing correct parallel programs may be even more difficult than novice programmers realize. That said, some programmers are more successful at writing parallel programs than others.

It is rare that a language or system is better than another one on every point of comparison. Even taking this into account, it is difficult to design a study which makes the tradeoffs between different systems clear. Language-level abstractions can lead to a dramatic improvement in code elegance, clarity, and size. In many contexts, this difference may be more important that squeezing the last drop of performance out of a program.

3. CHALLENGES

There is a semantic ambiguity in measuring productivity in a programming language that makes languages and features difficult to compare. We outline several challenges with measuring programmer productivity.

Identify who should write parallel programs and target usability for those programmers.

Generally, PL/HCI research has distinguished between novices, expert users, and end users [MK09].[3] It is much easier to run a controlled experiment with novice users (*e.g.* in the context of teaching a course), and so this is what many researchers have done. However, controlled experiments with novice users have been criticized for their ecological validity [HC09]. At this juncture, it is unclear whether parallel programming ability is a necessary skill for all programmers, or can be a specialization held by one member of a software development team.

It may be fruitful to categorize parallel programmers based on problem domain instead of experience level. For example, scientists may have different needs in a parallel programming system than application developers. One challenge is to collect information about types of parallel programmers, and then target subsequent usability research towards a particular group. Alternatively, given that few professional programmers are experts in parallel programming, perhaps strategies that have been successful for end users could be adapted to help programmers produce correct and scalable parallel code. A related issue is how to tell when parallel programmers have reached expert status. What set of questions or problems can be used to judge parallel programming expertise?

Software maintenance tasks are a standard part of the programming process.

Once the type of users to measure has been established, there is still a question of what it means to be productive for those users. Most usability analyses of programming systems involve participants writing stand-alone programs. This does not clearly extend to real world cases with large programs, legacy code, and different program styles. We feel that more research needs to focus on the usability of programming systems in terms of software maintainability; successful systems will make it more difficult for programmers to introduce bugs while modifying code. Future usability studies should investigate users debugging parallel programs, and users adding more parallel code to an existing code base.

Correctness is important.

It is appropriate to measure the difficulty of speeding up code while also gathering metrics pertaining to program correctness. Checking correctness of a program requires a clear specification, regression test suite, model checking system, etc. Collecting these necessary parts is more difficult for parallel programs [LPSZ08]. A correctness index of how easy it is to introduce bugs with different parallel programming systems would increase our understanding of the tradeoffs.

As distributed applications continue to scale up to larger systems, performance analysis must also consider an application's resilience in the face of failure [JDD08]. Checkpointing and restarting technology has been traditionally used in scientific simulations, but the overhead of this technique increases proportionally with the number of parts in the system. We feel that resilience to failure can have a large impact on the usability of a parallel programming system, but is difficult to quantify.

Benchmarks define the problem space.

Choosing good representative programming problems to measure is just as critical as choosing good metrics. Different paradigms may work better for different problems, and we still need clear productivity baselines [Gab96] across benchmarks. We believe that there will be no silver bullet parallel programming system that works for all types of problems. Pattern suites help define the problem space; examples include the Cowichan suite [Wil94] or the 13 Dwarfs [ABD+09]. One important area of future research is developing an agreed-upon taxonomy of core tasks and algorithms for a parallel

[3] End users are people who write programs but are not programmers [MK09].

programming system. Both microscopic and macroscopic views add valuable insight. Focusing solely on measuring small tasks could lead to optimizations that do not actually contribute a positive measurable effect in more complex problems, while focusing on any given larger application risks becoming irrelevant to other unrelated problems. For that reason, a weighted index of tasks and applications could provide the most generally relevant evaluation of the usability of different parallel programming methodologies.

We need better metrics for programmer productivity.

How can we measure programmer effort? Most studies which look beyond code performance attempt to measure the amount of time taken for participants to write a program. However, identifying which time was spent on coding is a nontrivial task. Some studies have asked participants to rate subjective impressions of difficulty or effort. A number of indirect metrics for effort have been used, including the number of LOC. Although LOC is recognized as an inadequate measure for effort [HCS+05], it has been used as a proxy for both programmer effort, under the assumption that shorter programs are faster to write (see, for example, [CYZEG04]), and also program correctness, under the assumption that shorter programs contain fewer bugs (see, for example, [SSE96]). It is unclear how subjective impressions or indirect metrics correlate with the performance of a program, or the amount of programmer time required to produce a program in the system or language. We need clearer metrics for describing programmer effort, and more powerful and reliable tools for gathering and evaluating such metrics [Luf09].

We would like metrics that will allow us to compare drastically different implementations of the same algorithm, and even different algorithms achieving the same goal. The following suggestions are provided as starting points for discussion. To be useful, these metrics will require more tools to help automate their collection and analysis.

Time-based metrics compare the amount of time to:

- create basic serial, optimized serial, basic parallel, and optimized parallel implementations using a particular programming system
- execute each implementation
- debug different classes of errors (*e.g.* computational kernel, parallelization, middleware, OS, hardware) using a particular programming system
- enhance an implementation with resilience features (checkpoint/restart, single and multiple component failures)
- document implementations
- perform a code review

Complexity metrics compare statistics collected over programs written with different systems:

- computer-assisted microanalysis [CW92, HCHP92]
- levels of nesting
- McCabe Cyclomatic Complexity [McC76]
- comparison of the numbers of various operations or characteristics in an implementation (*e.g.* bugs, tasks, memory transfers [Pro99])

We believe that metrics such as these will allow us to compare the ability of programmers when using different parallel programming paradigms. Understanding when different paradigms are most useful can help programmers more easily produce correct, high performance code.

4. FUTURE AREAS OF RESEARCH

Some directions for future research were outlined in the previous sections. A variety of additional questions surrounding usability and parallel programming remain to be answered. We here identify five additional topics: programming metaphors, visualization techniques, correctness comprehension, social support, and improved cost models.

Which metaphors are the most effective for explaining concepts of multithreading or other parallel computation? The importance of effective metaphors is often overlooked, but can have a significant impact on understanding, particularly for novice users of a system [LJ80, WT99, PM96]. What implicit assumptions do programmers make about different parallel programming paradigms?

What are effective visualization techniques for parallel programs? This is another relatively unexplored area within parallel programming research [gra09] that has usability implications. As the number of states and interactions between parts of a parallel program scale up, the gulf between detailed and global views expands. How can program visualizations improve the correctness and performance of parallel programs?

Do people that write parallel programs understand different correctness criterions for those programs? How can we make correctness conditions easier to understand? What predicts who can write correct parallel programs? Several prior usability studies [Luf09, ESEG+06] of parallel programming systems found that a significant group of participants were unable to produce a faster, correct solution to the programming problem posed.

Programmers work in teams, and the code they write will be passed on to later generations. What parallel programming system features can help communicate the meaning of a piece of code quickly and accurately to other team members, managers, or later programmers? Also, how can we build social support for writing parallel programs?

Achieving good parallel performance requires a substantial investment in time and effort to master various levels of abstraction, and often requires a solid understanding of the hardware/software stack. For example, false sharing can occur if two regions of memory share the same cache line but are used by different threads. However, detecting false sharing is difficult at the language level: a programmer needs to understand how caching behaviours affect their program. In other words, the language-level cost model is incomplete. We need a way to expose a more accurate cost model, without throwing away the abstractions we built.

5. CONCLUSION

Usability is a key factor in the effectiveness of a parallel programming system; we have outlined several future research directions on this topic. We need better metrics for programmer effort and parallel programming expertise, and a comprehensive taxonomy of parallel problems. We need to address program correctness as a usability issue, and understand who writes parallel programs. We need to continue traveling down this last mile of parallel programming and usability, because the future is already here.

6. REFERENCES

[ABD+09] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz,

N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, et al. A view of the parallel computing landscape. *Communications of the ACM*, 52(10):56–67, 2009.

[CCZ07] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 21:291–312, 2007.

[CW92] Jacques Cohen and Aline Weitzman. Software tools for micro-analysis of programs, 1992.

[CYZEG04] F. Cantonnet, Y. Yao, M. Zahran, and T. El-Ghazawi. Productivity analysis of the UPC language. In *IPDPS Workshop*, 2004.

[ES05] R. Eccles and D. A. Stacey. Understanding the parallel programmer. In *International Symposium on High Performance Computing Systems and Applications (HPCS)*, 2005.

[ESEG+06] K. Ebcioglu, V. Sarkar, T. El-Ghazawi, J. Urbanic, and P.S. Center. An experiment in measuring the productivity of three parallel programming languages. In *Workshop on Productivity and Performance in High-End Computing (PPHEC)*, 2006.

[FKGB10] Xiaoli Fern, Chaitanya Komireddy, Valentina Grigoreanu, and Margaret Burnett. Mining problem-solving strategies from HCI data. *ACM Transactions on Computer-Human Interaction*, 17(1):1–22, 2010.

[Gab96] Richard P. Gabriel. Productivity: Is there a silver bullet? In *Patterns of Software: Tales from the Software Community*. Oxford University Press, 1996.

[GBR10] Valentina I. Grigoreanu, Margaret M. Burnett, and George G. Robertson. A strategy-centric approach to the design of end-user debugging tools. In *Conference on Human factors In computing systems (CHI)*, 2010.

[gra09] Grand challenges in programming languages. In *Symposium on Principles of Programming Languages (POPL)*, 2009.

[HBVG08] L. Hochstein, V.R. Basili, U. Vishkin, and J. Gilbert. A pilot study to compare programming effort for two parallel programming models. *Journal of Systems and Software*, 81(11):1920–1930, 2008.

[HC09] C.A. Halverson and J. Carver. Climbing the Plateau: Getting From Study Design to Data That Means Something. In *Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU)*, 2009.

[HCHP92] Timothy J. Hickey, Jacques Cohen, Hirofumi Hotta, and Thierry PetitJean. Computer-assisted microanalysis of parallel programs. *Transactions on Programming Languages and Systems*, 14(1):54–106, 1992.

[HCS+05] L. Hochstein, J. Carver, F. Shull, S. Asgari, and V. Basili. Parallel programmer productivity: A case study of novice parallel programmers. In *High Performance Networking and Computing (SC)*, 2005.

[HPCa] High performance computing challenge award competition. `http://www.hpcchallenge.org/`.

[hpcb] High Productivity Computing Systems. `http://www.highproductivity.org/`.

[JDD08] William M. Jones, John T. Daly, and Nathan A. DeBardeleben. Application resilience: Making progress in spite of failure. *IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, 2008.

[Kus06] Bradley C. Kuszmaul. A Cilk response to the HPC challenge (class 2), 2006.

[Lee06] Edward A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[LJ80] G. Lakoff and M. Johnson. *Metaphors we live by*. University Of Chicago Press, 1980.

[LPSZ08] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[Luf09] M. Luff. Empirically Investigating Parallel Programming Paradigms: A Null Result. In *Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU)*, 2009.

[McC76] Thomas J. McCabe. A complexity measure. In *International Conference on Software Engineering (ICSE)*, 1976.

[MGM+09] P.E. McKenney, M. Gupta, M.M. Michael, P. Howard, J. Triplett, and J. Walpole. Is Parallel Programming Hard, And If So, Why? Unpublished, 2009.

[MK09] B.A. Myers and A.J. Ko. The past, present and future of programming in HCI. In *Human-Computer Interaction Consortium (HCIC)*, 2009.

[Nie] Jakob Nielsen. `http://www.useit.com/alertbox/20030825.html`.

[PM96] J.F. Pane and B.A. Myers. Usability issues in the design of novice programming systems. *CMU Human-Computer Interaction Institute Technical Report CMU-HCII-96-101*, 1996.

[Pro99] H. Prokop. Cache-oblivious algorithms, 1999.

[RHW10] C.J. Rossbach, O.S. Hofmann, and E. Witchel. Is transactional programming actually easier? In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2010.

[SSE96] D. Szafron, J. Schaeffer, and A. Edmonton. An experiment to measure the usability of parallel programming systems. *Concurrency Practice and Experience*, 8(2):147–166, 1996.

[Wil94] G.V. Wilson. Assessing the usability of parallel programming systems: The Cowichan problems. In *IFIP WG10. 3 Conference on programming environments for massively parallel distributed systems*, 1994.

[WT99] A. Whitten and J.D. Tygar. Why Johnny can't encrypt: A usability evaluation of PGP 5.0. In *USENIX Security Symposium*, 1999.