Are new languages necessary for multicore?

Edward A. Lee
Robert S. Pepper Distinguished Professor and Chair of EECS
EECS Department
University of California at Berkeley
Berkeley, CA 94720, U.S.A.
eal@eecs.berkeley.edu

Position Statement for Panel
2007 International Symposium on Code Generation and Optimization (CGO)
March 11-14, 2007, San Jose, California

February 28, 2007

It is widely acknowledged that concurrent programming is difficult. Yet multicore architectures make concurrent programming essential. If we understand why concurrent programming is so difficult, we have a better chance of solving the problem. Sutter and Larus observe [17]

> "humans are quickly overwhelmed by concurrency and find it much more difficult to reason about concurrent than sequential code. Even careful people miss possible interleavings among even simple collections of partially ordered operations."

Yet humans are actually quite adept at reasoning about concurrent systems. The physical world is highly concurrent, and our very survival depends on our ability to reason about concurrent physical dynamics. The problem is that we have chosen concurrent abstractions that do not even vaguely resemble the concurrency of the physical world. We have become so used to these computational abstractions that we have lost track of the fact that they are not immutable. I argue that the difficulty of concurrent programming is a consequence of the abstractions, and that if we are are willing to let go of those abstractions, then the problem will be fixable.

New abstractions for computing would seem to imply a need for new programming languages. However, this is not necessarily the case. I will argue that concurrency models can operate at the level of component architectures rather than programming languages. Indeed, if we augment object-oriented component models with intrinsic concurrency, very attractive programming models emerge. The models leverage existing languages and imperative reasoning about algorithms, and introduce concurrency via coordination of components rather than through modifications in the languages.

In general-purpose software engineering practice, one approach to concurrent programming dominates all others, namely, threads. Threads are sequential processes that share memory. They represent a key concurrency model supported by modern computers, programming languages, and operating systems. Many general-purpose parallel architectures in use today are direct hardware realizations of the thread abstraction.

Some applications can very effectively use threads. So-called "embarrassingly parallel" applications (for example, applications that essentially spawn multiple independent processes such as build tools, like PVM gmake, or web servers). Because of the independence of these applications, programming is relatively easy, and the abstraction being used is more like processes than threads (where memory is not shared). Where such applications do share data, they do so through database abstractions, which manage concurrency through such mechanisms as transactions. However, client-side applications are not so simple.

Of course, threads are not the only possibility for concurrent programming. In scientific computing, where performance requirements have long demanded concurrent programming, data parallel language extensions and message passing libraries (like PVM [8], MPI [14], and OpenMP[1]) dominate over threads for concurrent programming. In fact, computer architectures intended for scientific computing often differ significantly from so-called "general purpose" architectures. They commonly support vectors and streams in hardware, for example. However, even in this domain, concurrent programs remain tedious to write. C and FORTRAN dominate, despite a long history of much better data parallel languages.

In distributed computing, threads are often not a practical abstraction because creating the illusion of shared memory is often too costly. Even so, we have gone to considerable lengths to create distributed computing mechanisms that emulate multithreaded programming. CORBA and .NET, for example, are rooted in distributed object-oriented techniques, where software components interact with proxies that behave as if they were local objects with shared memory. Object-orientation's data abstraction limits the extent to which the illusion of shared memory needs to be preserved, so such techniques prove reasonably cost effective. They make distributed programming look much like multithreaded programming.

I have argued in [12] that nontrivial multithreaded programs are incomprehensible. The root of the problem is their wildly nondeterministic behavior due to arbitrary interleaving of atomic actions. I will argue that we must (and can) build concurrent models of computation that are far more deterministic, and that we must judiciously and carefully introduce nondeterminism only where needed. Threads take the opposite approach. They make programs absurdly nondeterministic, and rely on programming style to constrain that nondeterminism to achieve deterministic aims.

There are, of course, successful methods that greatly improve the usability of threads. Object-oriented design, for example, limits the visibility of data in software architectures, thereby limiting the effects of arbitrary interleaving. Transactions, which give programmers control over the granularity of atomic actions, also help enormously. Concurrent design patters, like MapReduce [6], and libraries of concurrent data structures, like those in Java 5.0 and STAPL [2], also help enormously. Language extensions can also help considerably, as for example in Split-C [5], Cilk [4], and Guava [3]. These language changes prune away considerable nondeterminacy without sacrificing much performance. More aggressive innovations, like *promises*[2] or futures [9] offer significantly different programming models, and may in fact catch on. Formal checkers, as for example in Blast [10] and the Intel thread checker[3], can help considerably by revealing program behaviors that are difficult for a human to spot. Less formal techniques, such as performance debuggers like Valgrind[4], can also help in a similar way, making it easier for programmers to sort through the vast nondeterminacy of

[1] See http://www.openmp.org

[2] See http://www.erights.org/

[3] See http://developer.intel.com/software/products/threading/tcwin

[4] See http://valgrind.org/

program behaviors.

While all of these techniques hold promise, I argue here for a different approach. I believe that the problem can be addressed by simply focusing on component architectures. Component architecture have become dominated by a particular view of object-oriented design as realized in C++, C#, and Java. This view is intrinsically imperative, where interactions between components are via call-return semantics, and threads and concurrency are not directly part of the component model. What flows through components is sequential control. An alternative model, where data flows through components (rather than control), has been called "actor-oriented" [13, 11, 1]. Such models can take many forms. Unix pipes offer an early form of actor-oriented design, and in fact were a very common concurrent programming model before threads emerged into the programmer's vernacular.

While actor-oriented design can be accomplished with new programming languages that replace imperative models, this is probably neither advisable nor necessary. I believe that the right answer is coordination languages. Coordination languages may introduce new syntax, but that syntax serves purposes that are orthogonal to those of established programming languages. Whereas a general-purpose concurrent language has to include syntax for mundane operations such as arithmetic expressions, a coordination language need not specify anything more than coordination. Given this, the syntax can be noticeably distinct. Coordination languages have been around for some time [15], and have failed to take root. One reason for this that their acceptance amounts to capitulation on one key front: homogeneity. A prevailing undercurrent in programming languages research is that any worthy programming language must be general purpose. It must be, at a minimum, sufficiently expressive to express its own compiler. And then, adherents to the language are viewed as traitors if they succumb to the use of another language. Language wars are religious wars, and few of these religions are polytheistic.

A key development, however, has broken the ice. UML, which is properly viewed as a family of languages, each with a visual syntax, is routinely combined with C++ and Java. Programmers are starting to get used to using more than one language, where complementary features are provided by the disjoint languages. There are many challenges on this path. Designing good coordination languages is no easier than designing good general-purpose languages, and is full of pitfalls. And of course, coordination languages need to develop scalability and modularity features analogous to those in established languages. This can be done. Our own Ptolemy II [7], for example, provides a sophisticated, modern type system at the coordination language level [18], and offers a preliminary form of inheritance and polymorphism that is adapted from object-oriented techniques [13]. A huge opportunity exists in adapting the concept of higher-order functions from functional languages to coordination languages to offer constructs like MapReduce at the coordination language level. Some very promising early work in this direction is given by Reekie [16].

Concurrency in software is difficult. However, much of this difficulty is a consequence of the abstractions for concurrency that we have chosen to use. The dominant one in use today for general-purpose computing is threads. But non-trivial multi-threaded programs are *incomprehensible to humans*. It is true that the programming model can be improved through the use of design patterns, better granularity of atomicity (e.g. transactions), improved languages, and formal methods. However, these techniques merely chip away at the unnecessarily enormous nondeterminism of the threading model. The model remains intrinsically intractable.

If we expect concurrent programming to be mainstream, and if we demand reliability and predictability from programs, then we must discard threads as a programming model. Concurrent

programming models can be constructed that are much more predictable and understandable than threads. Threads must be relegated to the engine room of computing, to be suffered only by expert technology providers.

1 Acknowledgements

I would like to acknowledge thought-provoking comments and suggestions from Joe Buck (Synopsys), Mike Burrows (Google), Stephen Edwards (Columbia), Jim Larus (Microsoft), and Sandeep Shukla (Virginia Tech).

References

[1] G. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. The MIT Press Series in Artificial Intelligence. MIT Press, Cambridge, MA, 1986.

[2] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. STAPL: An adaptive, generic parallel C++ library. In *Wkshp. on Lang. and Comp. for Par. Comp. (LCPC)*, pages 193–208, Cumberland Falls, Kentucky, 2001.

[3] D. F. Bacon, R. E. Strom, and A. Tarafdar. Guava: a dialect of Java without data races. In *ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, volume 35 of *ACM SIGPLAN Notices*, pages 382–400, 2000.

[4] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP)*, ACM SIGPLAN Notices, 1995.

[5] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. v. Eicken, and K. Yelick. Parallel programming in Split-C. In *Supercomputing*, Portland, OR, 1993.

[6] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Sixth Symposium on Operating System Design and Implementation (OSDI)*, San Francisco, CA, 2004.

[7] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity—the Ptolemy approach. *Proceedings of the IEEE*, 91(2), 2003.

[8] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine — A Users Guide and Tutorial for Network Parallel Computing*. MIT Press, Cambridge, MA, 1994.

[9] J. Henry G. Baker and C. Hewitt. The incremental garbage collection of processes. In *Proceedings of the Symposium on AI and Programming Languages*, volume 12 of *ACM SIGPLAN Notices*, pages 55–59, 1977.

[10] T. A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In *15th International Conference on Computer-Aided Verification (CAV)*, volume 2725 of *Lecture Notes in Computer Science*, pages 262–274. Springer-Verlag, 2003.

[11] C. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artifical Intelligence*, 8(3):323363, 1977.

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

[13] E. A. Lee and S. Neuendorffer. Classes and subclasses in actor-oriented design. In *Conference on Formal Methods and Models for Codesign (MEMOCODE)*, San Diego, CA, USA, 2004.

[14] Message Passing Interface Forum. MPI2: A message passing interface standard. *International Journal of High Performance Computing Applications*, 12(1-2):1–299, 1998.

[15] G. Papadopoulos and F. Arbab. Coordination models and languages. In M. Zelkowitz, editor, *Advances in Computers - The Engineering of Large Systems*, volume 46, pages 329–400. Academic Press, 1998.

[16] H. J. Reekie. Toward effective programming for parallel digital signal processing. Ph.D. Thesis Research Report 92.1, University of Technology, Sydney, 1992.

[17] H. Sutter and J. Larus. Software and the concurrency revolution. *ACM Queue*, 3(7), 2005.
[18] Y. Xiong. An extensible type system for component-based design. Ph.D. Thesis Technical Memorandum UCB/ERL M02/13, University of California, Berkeley, CA 94720, May 1 2002.