Panel
Multicore, Manycore, and Cloud Computing: Is a New Programming Language Paradigm Required?

S. Tucker Taft
SofCheck, Inc.
Burlington, MA
stt@sofcheck.com

Joshua Bloch
Google, Inc.
josh@bloch.us

Robert Bocchino
Institute for Software Research
Carnegie Mellon University
rbocchin@cs.cmu.edu

Sebastian Burckhardt
Microsoft Research
Redmond, WA
sburckha@microsoft.com

Hassan Chafi
Oracle Labs and
Pervasive Parallelism Laboratory
Stanford University
hchafi@stanford.edu

Russ Cox
Google, Inc.
rsc@google.com

Benedict Gaster
AMD, Inc.
benedict.gaster@amd.com

Guy Steele
Oracle Research
guy.steele@oracle.com

David Ungar
IBM Research
davidungar@ibm.com

Abstract

Most of the mainstream programming languages in use today originated in the 70s and 80s. Even the scripting languages in growing use today tend to be based on paradigms established twenty years ago. Does the arrival of multicore, manycore, and cloud computing mean that we need to establish a new set of programming languages with new paradigms, or should we focus on adding more parallel programming features to our existing programming languages?

Consistent with the SPLASH theme of the *Internet as the world-wide Virtual Machine*, and the *Onward!* theme focused on the future of *Software Language Design*, this panel will discuss the role that programming languages should play in this new distributed, highly parallel computing milieu. Do we need new languages with new programming paradigms, and if so, what should these new languages look like?

Categories and Subject Descriptors D.1.3 [**Programming Techniques**]: Concurrent Programming – distributed programming, parallel programming; D.3.2 [**Programming Languages**]: Language Classifications -- concurrent, distributed, and parallel languages; D.3.3 [**Programming Languages**]: Language Constructs and Features – concurrent programming structures.

General Terms Algorithms, Performance, Languages,.

Keywords multicore programming; manycore programming; cloud computing; new programming paradigms.

SPLASH'11 Companion, October 22–27, 2011, Portland, Oregon, USA.
ACM 978-1-4503-0940-0/11/10.

1. Introduction

Over the past decade, there have been various approaches to addressing the challenge represented by the explosion in parallel and distributed systems. In the programming language world, two fundamentally different approaches have been taken, namely designing new languages specifically adapted to this new challenge, versus extending existing languages with new parallel or distributed programming features. On this panel we have aficionados of both approaches, sometimes residing within the same person.

New languages of interest to this panel include Clojure, Go, Fortress, ParaSail, and Scala. Each of these languages incorporates one or more fundamentally new programming paradigms intended to address the challenges of parallel or distributed programming. Extensions of existing languages of interest to this panel include Cilk+, CUDA, Deterministic Parallel Java (DPJ), MPI, and OpenCL. These languages start from an existing language, typically C, C++, or Java, and then add in additional libraries and/or language features that add capabilities without restricting the use of the existing language features.

This panel will discuss the merits and costs associated with these two approaches to addressing the parallel and distributed computing environment challenge, and provide specific examples of how these distinct approaches can help the programmer navigate this new sea of possibilities.

2. Panel Members

2.1 S. Tucker Taft *(panel organizer)*

S. TUCKER TAFT is Founder and CTO of SofCheck, Inc., a company devoted to providing tools and technologies for helping to improve software quality and increase programmer productivity. From 1990 to 1995, Mr. Taft served as the lead designer of the Ada 95 programming language. In 2001, he led the architecture and development effort of the J2EE-based Mass.gov portal for the Commonwealth of Massachusetts. In 2002, Mr. Taft founded SofCheck. From 2001 to the present, Mr. Taft has been a member of the ISO Rapporteur Group that developed Ada 2005, and more recently is finalizing Ada 2012. In September 2009 Mr. Taft embarked on the design of ParaSail, Parallel Specification and Implementation Language, a new language that marries pervasive parallelism with formal methods.

New languages with new paradigms are critical to meeting the challenge of multicore and manycore computing. Programmers using existing languages are already nearing the limit of what they can manage in the way of complexity. Adding in the need to take full advantage of the exponentially increasing number of processors per chip, without new programming paradigms, will likely push past what most software development organizations can handle. Developing large software systems will take longer and longer, and may never reach the level of quality and stability required for the given application domain.

ParaSail represents an attempt to create a new language with relatively familiar syntax and semantics, but with some critical differences in terms of control structuring (the language has a pervasively parallel execution model), data structuring (no pointers, global variables, or aliasing), and error checking (all error checking is performed at compile-time, including all race-condition checking, uninitialized or null data reference, array index out-of-bounds, numeric overflow, etc.). Formal annotations, such as preconditions, postconditions, and invariants, are woven into the core syntax and enforced at compile time. Initial experience with this combination shows the language to have a profound effect on the parallelism and robustness of the code, while still being straightforward for existing programmers to learn and to use effectively.

2.2 Josh Bloch

JOSH BLOCH is Chief Java Architect at Google, author of the bestselling, Jolt Award-winning "Effective Java" (Addison-Wesley, 2001; Second Edition, 2008), and coauthor of "Java Puzzlers: Traps, Pitfalls, and Corner Cases" (Addison-Wesley, 2005) and "Java Conurrency in Practice" (Addison-Wesley, 2006). He was previously a Distinguished Engineer at Sun Microsystems, where he led the design and implementation of numerous Java platform features including the Java Collections Framework and JDK 5.0 language enhancements. He holds a Ph.D. from CMU and a B.S. from Columbia.

In a recent Wall Street Journal article entitled "Why Software Is Eating The World," Marc Andreessen said: "All of the technology required to transform industries through software finally works and can be widely delivered at global scale." I agree, but I would add: "so long as programmers are willing to hold their noses and assemble an odd assortment of barely-functional software with duct tape and baling wire." So the question is, will we improve on this state of affairs, or will worse-is-better design (as described by Dick Gabriel) carry the day?

The rise of multicore processors and cloud computing certainly does present a great opportunity for language and library designers. We've had a string of modest successes, such as Doug Lea's java.util.concurrent, and Jeff Dean and Sanjay Ghemawat's MapReduce. These systems solve real problems and make the world a better place. Can we do better? Is there a great new paradigm waiting to be discovered, or do pragmatic concerns favor evolutionary approaches? Others have opined that such a paradigm already exists (Actors, Transactional Memory, and Functional Programming come to mind). While these paradigms all have their uses, I'm reasonably certain that none of them are the silver bullet for making use of the new generation of multicore hardware.

So where is this all leading? In the words of the Magic 8-Ball, "the future is cloudy." Yes, the opportunity for a distinctly better new platform aimed squarely at cloud computing and multicore hardware is staring us in the face. But the inertia of existing platforms has never been greater. If I had to place my bets, I'd guess we're stuck with evolutionary improvements for the next decade, with the possibly of something fundamentally new and different thereafter. But I've never been all that good at prognosticating.

2.3 Robert Bocchino

ROBERT BOCCHINO is a Postdoctoral Associate at Carnegie Mellon University. His research interests lie in programming language design, type theory, formal verification, and

concurrency. Robert completed his Ph.D. at the University of Illinois at Urbana-Champaign in fall 2010. His dissertation described a Java-based object-oriented parallel language called Deterministic Parallel Java (DPJ). DPJ uses a novel effect system to (1) guarantee that parallel programs execute deterministically unless nondeterminism is explicitly requested; (2) ensure that any nondeterminism is subject to strong safety guarantees, including freedom from data races; and (3) check that uses of object-oriented parallel frameworks are safe. At CMU, Robert is working with Jonathan Aldrich on the Plaid programming language and on designing and verifying high-level abstractions that make it easy for programmers to write correct and efficient parallel code.

Writing correct and efficient parallel or distributed code is hard. So the increasing prevalence of parallel and cloud computing poses challenges for both programmer productivity and code quality. I believe that the most important way to meet these challenges is to develop good abstractions. Ideally, a programmer working in a particular domain should use domain-specific abstractions, with the details of parallel correctness and efficiency hidden in the implementation. That way, a parallelism expert can write and tune the implementation, with most programmers just interacting with the API. Most programmers should not be writing low-level code with threads, locks, and communication primitives, though such code might be hidden in the implementation.

In my view, once we get the abstractions right, whether we present them as a new language or as an extension of an existing language is a secondary consideration. Modern languages like C++, Java, and C# have sufficiently powerful type systems that these kinds of abstractions can be presented as libraries or frameworks. New languages can have significant benefits: their design can draw on both new ideas in language design and experience with features that worked well or didn't in previous languages. And a special-purpose language may be able to express programming concepts more cleanly or efficiently than a library or framework. Where the investment in tool chain infrastructure is warranted, new languages can be developed. But the most important thing is to get the abstractions right.

2.4 Sebastian Burckhardt

SEBASTIAN BURCKHARDT was born and raised in Basel, Switzerland, and studied Mathematics at the local University. During an exchange year at Brandeis University, he discovered his affinity to Computer Science and immigrated to the United States, where he completed a second Master's degree. After a few years of industry experience at IBM, he returned to academia and earned his PhD in Computer Science at the University of Pennsylvania. For the past 4 years, he has worked as a researcher at Microsoft Research in Redmond. His research interests revolve around the general problem of programming concurrent, parallel, and distributed systems conveniently, efficiently, and correctly. More specific interests include memory consistency models, concurrency testing, self-adjusting computation, and the concurrent revisions programming model.

In my opinion, the biggest challenge and opportunity for PL research today is to discover understandable, robust, and efficient abstractions that empower programmers to work seamlessly with shared state in a world where concurrency, parallelism, and distribution are omnipresent. On multicores, we need programming languages to rise to an abstraction level where correctness issues (such as atomicity violations, data races, and deadlocks) and performance concerns (such as scheduling) are no longer nasty surprises, but become transparent properties of a program. In browser or mobile applications, we want languages to support the illusion of a consistent, persistent, global, shared state; specifically, we need abstractions that reduce the amount of explicit copying and conflict management by the programmer, but do so without introducing expensive scalability bottlenecks.

In either case, the question of whether we need a new language or extend existing languages is somewhat secondary. Once we find the right abstractions, they can provide tremendous value, whether they be delivered as libraries, language extensions, or domain-specific languages.

2.5 Hassan Chafi

HASSAN CHAFI is a research manager at Oracle Labs leading a couple of projects in the area of heterogeneous computing. Hassan is in the final stages of pursuing a PhD degree at Stanford University. He is being advised by Kunle Olukotun. Some of Hassan's initial work at Stanford included implementing a profiling environment for programs using Transactional Memory and implementing a scalable version of the Transactional Coherence and Consistency (TCC) protocol. After taking a leave of absence during which he founded GenieTown, a start-up with the goal of creating an online marketplace for local services he returned to Stanford to join the newly formed Pervasive Parallelism Laboratory (PPL). At PPL, Hassan has been leading a team of talented graduate students working on building Delite, an open-source infrastructure to support the implementation and execution of performance oriented DSLs.

A Domain-Specific Approach to Multicore, Manycore and Cloud Computing: Ideally a parallel programming language should provide generality, high productivity, and produce high-performance binaries that take advantage of all the hardware resources available in a given platform. Unfortunately, no such language currently exists and some of these goals seem to be conflicting. Most successful languages usually focus on only two of the aforementioned goals. For example C/C++ and Java are high performance and general, but not highly productive. Python and Ruby are highly productive and general but suffer from poor performance.

We propose focusing on high productivity and performance by foregoing generality. This can be achieved via the use of Domain-Specific Languages (DSLs). There are already a few examples of successful DSLs in widespread use (SQL, OpenGL, Matlab). DSLs have a long history of increasing programmer productivity by providing extremely high-level, in a sense "ideal", abstractions tailored to a particular domain. DSLs are usually declarative in nature which leads to programs that express intent as opposed to implementation details. Such details are best left to a compiler. Since the programs are expressed at a very high level, a DSL compiler can generate different concrete implementations suited to the various execution environments (Multicore, GPU, Cluster or a hybrid). The Pervasive Parallelism Laboratory (PPL) at Stanford has been working diligently on lowering the bar for the implementation and adoption of such performance oriented DSLs. We will be presenting some of the insights uncovered so far by the PPL and discuss the big challenges that remain to be solved.

2.6 Russ Cox

RUSS COX worked on the open source releases of Plan 9 from Bell Labs and wrote the search engine for the Online Encyclopedia of Integer Sequences before joining Google. At Google, he led the design and implementation of Google Code Search, a regular expression-based search engine for public source code, and has for the last three years worked on Go. He holds bachelor's and master's degrees in computer science from Harvard University and a Ph.D. in computer science from M.I.T.

The paradigms needed to address the challenges of multicore, manycore, and cloud computing are not new, but neither are they embodied in the mainstream programming languages used today. The most widespread paradigm for multicore computing today is threads sharing memory and synchronizing with mutexes and condition variables. Programmers who can write correct, scalable programs with these primitives are the exception, not the rule.

The Go programming language (golang.org) turns to an alternate approach, shared memory coordinated by message passing in the style of Hoare's CSP. That is, instead of communicating by sharing memory, Go programs share memory by communicating. We have found that programmers new to concurrent and parallel programming are much more likely to write correct programs when reasoning about communication instead of synchronization. This explicit focus on communication also generalizes well to thinking about the cloud.

2.7 Benedict Gaster

BENEDICT GASTER is an architect at AMD, where he is working on programing models for heterogeneous APUs containing multicore and manycore architectures. He has a PhD in computer science for his work on type systems for records and variants.

The move towards Multicore, Manycore, and even more recently Cloud Computing has again pushed the envelope in computer science for the development of new programming languages and paradigms but is this really what's needed? There are too many programing languages to mention, each of which have their advantages and disadvantages, but many of them lack concurrency abstractions and ones that do have them often emit a strong model for reasoning about communication and shared access to data. The last few years have seen an effort to define stronger memory models for shared memory programming with respect to data race free (DRF) programs, c.f. Java and C++11, and with this direction comes hope. Not because DRF shared memory programming is necessarily the final answer but because it looks beyond a particular programming language or paradigm and instead tries to address the fundamentals of building software for Multicore and Manycore machines, by providing a clear semantics for when and how data can be shared with others.

2.8 Guy Steele

GUY STEELE has spent most of his career as a slacker, only managing to accomplish:

- Contributions to and illustrations for *The Hacker's Dictionary*
- Design of the original Emacs command set
- The first port of TeX
- *C: A Reference Manual* (with Sam Harbison) and standards committee work for C
- *Common Lisp: The Language* which became the ANSI standard for Common Lisp
- Definition of the Scheme dialect of Lisp (with Gerald Sussman)
- The "Lambda Papers" (with Gerald Sussman)

- *The_High Performance Fortran Handbook* (with Koelbel, Loveman, Schreiber, and Zosel) and standards committee work for Fortran and High Performance Fortran
- *The Java Language Specification*
- Project editor for the first ECMAScript standard (aka Javascript)
- Currently working on Fortress, a growable parallel language that integrates object-oriented and mathematical notation

(Bio by Alex Miller, slightly updated to give credit where it's due, this sentence being one example.)

Not only is a new programming language paradigm required, but in fact more than one is required.

If we consider the famous "Fallacies of Distributed Computing" (enumerated by Peter Deutsch, with input from Bill Joy, Tom Lyon, and James Gosling — see http://en.wikipedia.org/wiki/Fallacies_of_Distributed_Computing) we see that both multicore applications and cloud applications need to take their principles into account in ways that existing popular languages do not—but we also see that there are tradeoffs, and principles that are less important for multicore computing become much more important for cloud computing. These tradeoffs have implications for language design.

In addition, we argue that existing language designs encourage programming strategies that are inherently too difficult to parallelize automatically; new language designs need to encourage the expression of divide-and-conquer algorithms rather than algorithms that rely on iterative accumulation of results.

2.9 David Ungar

DAVID UNGAR is an out-of-the-box thinker who enjoys the challenge of building computer software systems that work like magic and fit a user's mind like a glove. He received the 2009 Dahl-Nygaard award for outstanding career contributions in the field of object-orientation, and was honored as an ACM Fellow in 2010. Three of his papers have been honored by the Association for Computing Machinery for lasting impact over ten to twenty-four years: for the design of the prototype-based Self language, dynamic optimization techniques, and the application of cartoon animation ideas to user interfaces. He enjoys a position at IBM Research, where he is taking on a new challenge: investigating how application programmers can exploit manycore systems, and testing those ideas to see if they can help with large-scale data analysis.

In the not-too-distant future, manycore microprocessors will become commonplace: every (non-hand-held) computer's CPU chip will contain 1,000 fairly homogeneous cores. Such a system will not be programmed like the cloud, or even a cluster because communication will be much faster relative to computation. Nor will it be programmed like today's multicore processors because the illusion of instant memory coherency will have been dispelled by both the physical limitations imposed by the 1,000-way fan-in to the memory system, and the comparatively long physical lengths of the inter- vs. intra-core connections. When this future arrives we will have to change our very model of computation.

If we cannot skirt Amdahl's Law, the last 900 cores will do us no good whatsoever. What does this mean? We cannot afford even tiny amounts of serialization. Locks?! Even lock-free algorithms will not be parallel enough. They rely on instructions that require communication and synchronization between cores' caches. We need to develop a body of knowledge around computing without any synchronization whatsoever.

In our Renaissance project at IBM, Vrije, and Portland State, (http://soft.vub.ac.be/~smarr/renaissance/) we are investigating what we call "anti-lock," "race-and-repair," or "end-to-end nondeterministic" computing. When we give up synchronization, we of necessity give up determinism. (In fact, there seems to be a fundamental tradeoff between determinism and performance, but that's another topic.) The obstacle we shall have to overcome, if we are to successfully program manycore systems, is our cherished assumption that we write programs that always get the exactly right answers. This assumption is deeply embedded in how we think about programming. The folks who build web search engines already understand, but for the rest of us, to quote Firesign Theatre: **Everything You Know Is Wrong!**

Ungar will be expanding on this topic in an invited talk at the Dynamic Languages Symposium 2011 co-located with the SPLASH 2011 conference.