OOPSLA Wkshp. on Curricula for Concurrency, Orlando, FL, Oct. 2009.

Making the Simple Case Simple

Michael L. Scott
University of Rochester

October 2009

With the proliferation of multicore processors, and the anticipation of "many-core," it has become commonplace to predict dire consequences if researchers fail to make it easy for rank-and-file programmers to write correct, efficient parallel code. Certainly anyone who has taught traditional concurrency knows that students have great difficulty with the subject. Much of the pedagogical problem, I believe, stems from a tendency to introduce material in the way it developed historically, starting with data races and the implementation of mutual exclusion (and today including, by necessity, memory consistency models).

In the "real world," only concurrency wizards introduce low-level data races into their code on purpose. Above the level of wizards, there would seem to be several "tiers" of parallel programming techniques, with progressively less conceptual complexity (Table 1). One tier up from the bottom of the table, explicitly synchronized concurrency is a mature and well-understood discipline, though still generally too complex for most programmers. Transactional memory (TM) may simplify matters considerably, but will not be a panacea: it does not eliminate the need to think about high-level races. Ubiquitous parallelism, effected by "ordinary" programmers, will likely require innovation in Tiers 1 and 2, where even high-level races are precluded, and program behavior cannot change due to changes in thread interleaving.

In a recent article, Bocchino et al. [2] argue eloquently for a *deterministic* model of parallel programming, in which the introduction of concurrency does not change program behavior. Similar arguments have been made by many others, going back at least as far as Multilisp [5]. Potential approaches include automatic parallelization of (mostly loop-based) programs; side-effect-free programming; *safe futures* in imperative languages [14]; pipelining [7]; partially ordered iterators [6]; ordered locking [10]; and data partitioning effected with types [1]. Unfortunately, the longevity of the argument, the diversity of approaches, the flurry of recent papers, and the complexity of some proposed solutions all make deterministic parallelism look more like a vibrant research topic than an established technology suitable for undergraduate instruction. So what should we be doing in the classroom?

I suggest 4 concrete principles:

(1) Make it easy to express concurrent tasks. Algol 68 had simpler concurrency syntax (`begin A, B end`) than any language in widespread use today, with the possible exception of Fortran 95. We need equally simple syntax in modern teaching languages. Calling two independent subroutines in parallel—or mapping a pure function across the elements of a collection—should not require the import of special libraries, the instantiation of special classes, or the creation of closure objects.

(2) Teach independence as a programming discipline. Early in the curriculum, we need to establish the assumption that parallelism exists, by default, for the execution of mutually independent tasks, and we need to help students learn to build such tasks into their code. Once the habit is ingrained (later in the curriculum), we can relax the model a bit—e.g., to allow parallel tasks to make atomic calls to commutative operations.

(3) Leverage speculation as an implementation technique. As many researchers have observed, speculation can enable optimistic parallel execution in semantically sequential code, without changing the programming model. Sections of code worth executing speculatively may be identified by programmer hints, static analysis, or profile-driven feedback. Speculation may be implemented using virtual memory protection [4], software instrumentation [8, 9, 12], or hardware support [11, 13]. Deeply embedded in standard run-time systems, speculation can also support race detection in parallel code, transactional memory, and statically unsafe optimization.

(4) Embrace race detection as a debugging tool. Someday, we may all program in a language that is side effect free, or that makes it easy to partition the heap dynamically among concurrent tasks. Until then, students will continue to write code containing accidental data races, and we will need to help them find and eliminate those races. I lean toward dynamic detection, which can be retrofitted into existing languages without little or no modification, generates no false alarms, and can find all dynamic races at relatively modest cost (about 3× for typical code, using the latest instrumentation techniques from transactional memory [3]). At that cost, programmers will

Tier	Techniques	Issues	Curricular location
(1) Automatic	canned libraries, **parallelizing compilers**, VHL primitives	(none)	computer appreciation
(2) Deterministic	**do-all** or **futures**, with system-enforced independence or fall-back to sequential execution; vector for-all; pipelining; partially-ordered iterators	locality, granularity, load balance, design patterns	data structures
(3) Explicitly synchronized (data-race-free): (a) concurrent (event-driven); (b) parallel (thread-based); (c) distributed (message-based)	**atomic blocks (transactional memory)**, locks/monitors, loop post-wait, map/reduce, condition synchronization, **run-until-block**, send/receive/rendezvous	progress, consensus, happens-before, data-race freedom, 2-phase commit, self-stabilization, Byzantine agreement	(a) graphics, HCI, web computing; (b) SW engg., languages, scientific computing; (c) networks, distributed computing
(4) Low-level (racy)	implementation of threads and synchronization, nonblocking data structures	memory models, consistency, linear/serial-izability, consensus hierarchy	OS, architecture, parallel computing, DBMS

Table 1: Tiers of parallelism. Speculation may help to enable the techniques shown in **bold**.

typically want to turn checking off "once debugging is complete." While this is not ideal, it's hardly a new phenomenon, and emerging speculative hardware or language advances will eventually reduce it.

Guided by these principles, I believe we should introduce parallelism not as a special topic, but as one of the standard control flow mechanisms in the programmer's tool chest, along with selection, iteration, recursion, and the like. We should then introduce assignments—throughout the undergraduate curriculum—in which parallelism makes a difference, and we should run those assignments on multicore machines: students need the positive reinforcement of concrete, significant speedups when they parallelize their code.

References

[1] R. L. Bocchino Jr., V. S. Adve, D. Dig, S. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A Type and Effect System for Deterministic Parallel Java. *OOPSLA 2009*.

[2] R. Bocchino Jr., V. S. Adve, S. V. Adve, and M. Snir. Parallel Programming Must Be Deterministic By Default. *HotPar 2009*.

[3] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: Streamlining STM by Abolishing Ownership Records. *PPoPP 2010* (to appear).

[4] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software Behavior Oriented Parallelization. *PLDI 2007*.

[5] R. H. Halstead, Jr. Multilisp: A Language for Concurrent Symbolic Computation. *ACM TOPLAS*, 7(4):501-538, Oct. 1985.

[6] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic Parallelism Requires Abstractions. *PLDI 2007*.

[7] E. A. Lee. The Problem with Threads. *Computer*, 39(5):33-43, May 2006.

[8] M. Mehrara, J. Hao, P.-C. Hsu, and S. A. Mahlke. Parallelizing Sequential Applications on Commodity Hardware using a Low-cost Software Transactional Memory. *PLDI 2009*.

[9] C. E. Oancea, A. Mycroft, and T. Harris. A Lightweight In-Place Implementation for Software Thread-Level Speculation. *SPAA 2009*.

[10] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient Deterministic Multithreading in Software. *ASPLOS 2009*.

[11] M. K. Prabhu and K. Olukotun. Using Thread-level Speculation to Simplify Manual Parallelization. *PPoPP 2003*.

[12] M. F. Spear, K. Kelsey, T. Bai, L. Dalessandro, M. L. Scott, C. Ding, and P. Wu. Fastpath Speculative Parallelization. *LCPC 2009*.

[13] C. von Praun, L. Ceze, and C. Caşcaval. Implicit Parallelism with Ordered Transactions. *PPoPP 2007*.

[14] A. Welc, S. Jagannathan, and A. L. Hosking. Safe Futures for Java. *OOPSLA 2005*.