Future
The Problem with Threads

Edward A. Lee
University of California, Berkeley

For concurrent programming to become mainstream, we must discard threads as a programming model. Nondeterminism should be judiciously and carefully introduced where needed, and it should be explicit in programs.

Concurrent programming is difficult,[1] yet many technologists predict the end of Moore's law will be answered with increasingly parallel computer architectures—multicore or chip multiprocessors (CMPs).[2] If we hope to achieve continued performance gains, programs must be able to exploit this parallelism.

Automatic exploitation of parallelism in sequential programs, through either computer architecture techniques such as dynamic dispatch or automatic parallelization of sequential programs,[3] offers one possible technical solution. However, many researchers agree that these automatic techniques have been pushed to their limits and can exploit only modest parallelism. Thus, programs themselves must become more concurrent.

Understanding why concurrent programming is so difficult can help us solve the problem. The physical world is highly concurrent, and our very survival depends on our ability to reason about concurrent physical dynamics. This reasoning doesn't extend to concurrent programs because we have chosen abstractions that do not even vaguely resemble the physical world's concurrency. We have become so used to these computational abstractions that we have forgotten they are not immutable. The difficulty of concurrent programming is a consequence of these abstractions, and if we can let go of them, the problem will be fixable.

THREADS

In general-purpose software engineering practice, we have reached a point where one approach to concurrent programming dominates all others—namely, threads, 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—such as symmetric multiprocessors—are direct hardware realizations of the thread abstraction.

Some applications can use threads very effectively—for example, so-called embarrassingly parallel applications that essentially spawn multiple independent processes such as build tools (PVM gmake) or Web servers. Given these applications' independence, programming is relatively easy and the abstraction being used is more like processes than threads. 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.

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—such as PVM, MPI, and OpenMP—dominate over threads for concurrent programming. 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.

Nondeterminism and Threads

From a fundamental perspective, threads are seriously flawed as a computation model. To wit,

Let $N = \{0, 1, 2, \ldots\}$ represent the natural numbers and $B = \{0, 1\}$ be the set of binary digits. Let B^* be the set of all finite sequences of bits, and $B^\omega = (N \rightarrow B)$ be the set of all infinite sequences of bits, each of which is a function that maps N into B. Further, let $B^{**} = B^* \cup B^\omega$, which we will use to represent the state of a computing machine, its potentially infinite inputs, and its potentially infinite outputs. Finally, let Q denote the set of all partial functions with domain and codomain B^{**}. Partial functions are functions that may or may not be defined on each element of their domain.

An imperative machine (A, c) is a finite set $A \subset Q$ of *atomic actions* and a *control function* $c : B^{**} \rightarrow N$. The set A represents the atomic actions, typically instructions, of the machine; the function c represents how instructions are sequenced. We assume that A contains one halt instruction $h \in A$ with the property that

$$\forall \; b \in B^{**}, h(b) = b$$

That is, the halt instruction leaves the state unchanged.

A *sequential program* of length $m \in N$ is a function $p : N \rightarrow A$, where

$$\forall \; n \geq m, p(n) = h$$

That is, a sequential program is a finite sequence of instructions tailed by an infinite sequence of halt instructions. The set of all sequential programs, which we denote P, is a countably infinite set.

An execution of this program is a thread. It begins with an initial $b_0 \in B^{**}$, which represents the initial state of the machine and the potentially infinite input, and for all $n \in N$,

$$b_{n+1} = p(c(b_n))(b_n) \tag{1}$$

Here, $c(b_n)$ provides the index into the program p for the next instruction $p(c(b_n))$. That instruction is applied to the state b_n to get the next state b_{n+1}. If for any $n \in N$, $c(b_n) \geq m$, then $p(c(b_n)) = h$ and the program halts in state b_n and the state henceforth never changes. If for all initial states $b_0 \in B$ a program p halts, then p defines a total function in Q. If a program p halts for some $b_0 \in B$, then it defines a partial function in Q.

We now get to the core appeal that sequential programs have. Given a program and an initial state, the sequence given by Equation 1 is defined. If the sequence halts, then the function computed by the program is defined. Any two programs p and p' can be compared and be found equivalent if they compute the same partial function. That is, they are equivalent if they halt for the same initial states, and for such initial states their final state is the same. Such a theory of equivalence is essential for any useful formalism. In this classical theory, programs that do not halt are all equivalent; this creates serious problems when applying the theory of computation to embedded software, where useful programs do not halt.

We lose these essential and appealing properties of programs when multiple threads are composed. Consider two multithreaded programs, p_1 and p_2, that execute concurrently. In this case, we replace Equation 1 with the following

$$b_{n+1} = p_i(c(b_n))(b_n), \text{ where } i \in \{1, 2\} \tag{2}$$

At each step n, either program can provide the next atomic action. Consider now whether we have a useful theory of equivalence. That is, given a pair of multithreaded programs (p_1, p_2) and another pair (p'_1, p'_2), when are these two pairs equivalent? A reasonable extension of the basic theory defines them to be equivalent if all interleavings halt for the same initial state and yield the same final state. The enormous number of possible interleavings makes it extremely difficult to reason about such equivalence except in trivial cases where, for example, the state B^{**} is partitioned so that the two programs are unaffected by each other's partition.

Even worse, given two programs p and p' that are equivalent when executed according to Equation 1, if they execute in a multithreaded environment we can no longer conclude they are equivalent. We must know about all other threads that might execute—something that may not itself be well-defined—and we would have to analyze all possible interleavings. We conclude that no useful theory of equivalence can be applied to threads.

The core abstraction of computation given by Equation 1, on which all widely used programming languages are built, emphasizes deterministic composition of deterministic components. Both the actions and their sequential composition are deterministic. Sequential execution is, semantically, function composition—a neat, simple model where deterministic components compose into deterministic results.

In distributed computing, threads are often not a practical abstraction because creating the illusion of shared memory often costs too much. 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 must be preserved, so such techniques prove reasonably cost effective. They make distributed programming look much like multithreaded programming.

Yet this argument is not a resurrection of the old shared-memory versus message-passing debate. Message passing can be made as nondeterministic and difficult to understand as threads. Conversely, shared memory can be used in deterministic and understandable ways—using data-parallel languages, for example. The argument here is against the use of nondeterministic mechanisms to achieve deterministic aims.

Embedded computing also exploits concurrency models other than threads. Programmable DSP architectures are often VLIW machines. Video signal processors often combine SIMD with VLIW and stream processing. Network processors provide explicit hardware support for streaming data. However, despite considerable innovative research, in practice, programming models for these domains remain primitive. Designers write low-level assembly code that exploits specific hardware features, combining this code with C code only where performance is noncritical.

For many embedded applications, reliability and predictability are far more important than expressiveness or performance. We can argue that this should be true in general-purpose computing, but that's a side argument. I contend that achieving reliability and predictability using threads is essentially impossible for many applications.

THREADS AS COMPUTATION

From a fundamental perspective, threads are seriously flawed as a computation model because they are wildly nondeterministic, as the "Nondeterminism and Threads" sidebar describes. The programmer's job is to prune away that nondeterminism. We have developed tools to assist in the pruning: Semaphores, monitors, and more modern overlays on threads offer the programmer ever more effective pruning. But pruning a wild mass of brambles rarely yields a satisfactory hedge.

To offer another analogy, a folk definition of insanity is to do the same thing over and over again and expect the results to be different. By this definition, we in fact require that programmers of multithreaded systems be insane. Were they sane, they could not understand their programs.

Moreover, implementing a multithreaded computation model is difficult. Witness, for example, the subtleties with the Java memory model, where even

```
public class ValueHolder {
    private List listeners = new LinkedList();
    private int value;
    public interface Listener {
        public void valueChanged(int newValue);
    }
    public void addListener(Listener listener) {
        listeners.add(listener);
    }
    public void setValue(int newValue) {
        value = newValue;
        Iterator i = listeners.iterator();
        while(i.hasNext()) {
            ((Listener)i.next()).valueChanged(newValue);
        }
    }
}
```

Figure 1. A Java implementation of the observer pattern, valid for one thread.

astonishingly trivial programs produce considerable debate about their possible behaviors.[4]

We must and can build concurrent computation models that are far more deterministic, and we must judiciously and carefully introduce nondeterminism where needed. Nondeterminism should be explicitly added to programs, and only where needed, as it is in sequential programming. Threads take the opposite approach. They make programs absurdly nondeterministic and rely on programming style to constrain that nondeterminism to achieve deterministic aims.

HOW BAD IS IT IN PRACTICE?

In practice, many programmers today write multithreaded programs that work. This appears to be a contradiction, but programmers can employ tools that prune away much of the nondeterminism.

Object-oriented programming, for example, limits the visibility that certain portions of a program have into portions of the state. This effectively partitions the state space into disjoint sections. Where programs do operate on shared portions of this state space, semaphores, mutual-exclusion locks, and monitors (objects with mutually exclusive methods) provide mechanisms that programs can use to prune away more of the nondeterminism. But in practice, these techniques yield understandable programs only for very simple interactions.

Consider the observer pattern,[5] a very simple and widely used design pattern. Figure 1 shows a Java implementation valid for a single thread. This shows two methods from a class where an invocation of the setValue() method triggers notification of the new value by calling the valueChanged() method of any objects that have been registered by a call to addListener().

The code in Figure 1 is not thread safe, however. That is, if multiple threads can call setValue() or addListener(), the listeners list could be modified while the iterator is iterating through the list, triggering an exception that will likely terminate the program.

```
public class ValueHolder {
    private List listeners = new LinkedList();
    private int value;
    public interface Listener {
        public void valueChanged(int newValue);
    }
    public synchronized void addListener(Listener listener) {
        listeners.add(listener);
    }
    public void setValue(int newValue) {
        List copyOfListeners;
        synchronized(this) {
            value = newValue;
            copyOfListeners = new LinkedList(listeners);
        }
        Iterator i = copyOfListeners.iterator();
        while(i.hasNext()) {
            ((Listener)i.next()).valueChanged(newValue);
        }
    }
}
```

Figure 2. A Java implementation of the observer pattern that attempts to be thread safe.

The simplest solution adds the Java keyword synchronized to each of the setValue() and addListener() method definitions. The synchronized keyword in Java implements mutual exclusion, turning instances of this ValueHolder class into monitors and preventing any two threads from being in these methods simultaneously. When the program calls a synchronized method, the calling thread attempts to acquire an exclusive lock on the object. If any other thread holds that lock, the calling thread stalls until the lock releases.

However, this solution is unwise because it can lead to deadlock. In particular, suppose we have an instance *a* of ValueHolder and an instance *b* of another class that implements the Listener interface. That other class can do anything in its valueChanged() method, including acquiring a lock on another monitor. If it stalls in acquiring that lock, it will continue to hold the lock on this ValueHolder object while stalled. Meanwhile, whatever thread holds the lock it is trying to acquire might call addListener() on *a*. Both threads are now blocked with no hope of becoming unblocked. This sort of potential deadlock lurks in many programs that use monitors.

Already, this rather simple design pattern is proving difficult to implement correctly. Consider the improved implementation shown in Figure 2. While holding a lock, the setValue() method copies the listeners list. Since the addListeners() method is synchronized, this avoids the concurrent modification exception that might occur with the code in Figure 1. Further, it calls valueChanged() out-

side of the synchronized block to avoid deadlock.

This code is still not correct, however. Suppose two threads call setValue(). One will set the value last, leaving that value in the object. But listeners might be notified of value changes in the opposite order and will conclude that the final value of the ValueHolder object is the wrong one.

This pattern can be made to work robustly in Java. Yet even this very simple and commonly used design pattern has required some rather intricate thinking about possible interleavings.

I speculate that most multithreaded programs have such bugs. I speculate further that the bugs have not proved to be major handicaps only because today's architectures and operating systems deliver modest parallelism.

The cost of context switching is high, so only a tiny percentage of the possible interleavings of thread instructions ever occur in practice. I conjecture that most multithreaded general-purpose applications are so full of concurrency bugs that—as multicore architectures become commonplace—these bugs will begin to show up as system failures.

This paints a bleak scenario for computer vendors: Their next-generation machines will become widely known as the ones on which many programs crash.

These same computer vendors advocate more multithreaded programming to provide the concurrency that can exploit the parallelism they would like to sell us. Intel, for example, has embarked on an active campaign to get leading computer science academic programs to put more emphasis on multithreaded programming. If they succeed, and programmers make more intensive use of multithreading, the next generation of computers will become nearly unusable.

FIXING THREADS BY MORE AGGRESSIVE PRUNING

Several approaches to solving this concurrency problem share a common feature. Specifically, they preserve the essential thread model of computation for programmers, but provide them with more aggressive mechanisms for pruning its enormously nondeterministic behavior.

Software engineering processes

Better software engineering processes provide the first technique. While essential for reliable multithreaded programs, these processes are not sufficient.

An anecdote from the Ptolemy Project is telling. In early 2000, my group began developing the Ptolemy II kernel,[6] a modeling environment supporting concurrent computation models. An early objective was to permit modification of concurrent programs via a graphical user interface while those programs executed. The challenge involved ensuring that no thread could ever see an inconsistent view of the program structure. The strategy used Java threads with monitors (http://ptolemy.eecs.berkeley.edu).

Part of the Ptolemy Project experiment sought to determine whether we could develop effective software engineering practices for an academic research setting. We developed a process that included a four-level code maturity rating system (red, yellow, green, and blue), design reviews, code reviews, nightly builds, regression tests, and automated code coverage metrics.[7] We wrote the kernel portion that ensured a consistent view of the program structure in early 2000, design reviewed to yellow, and code reviewed to green. The reviewers included concurrency experts, not just inexperienced graduate students.

We wrote regression tests that achieved 100 percent code coverage. The nightly build and regression tests ran on a two-processor SMP machine, which exhibited different thread behavior than the development machines, which all had a single processor.

The Ptolemy II system itself became widely used, and every use of the system exercised this code. No problems were observed until the code deadlocked in April 2004, four years later.

Our relatively rigorous software engineering practice had identified and fixed many concurrency bugs. But that a problem as serious as a deadlock could go undetected for four years despite this practice is alarming. How many more such problems remain? How long must we test before we can be sure to have discovered all such problems? Regrettably, I must conclude that testing might never reveal all the problems in nontrivial multithreaded code.

There are tantalizingly simple rules for avoiding deadlock, however. For example, always acquire locks in the same order.[8] However, this rule is difficult to apply because no method signature in any widely used programming language indicates what locks the method acquires. We must examine the source code of all methods called—and all methods that those methods call—to confidently invoke a method.

Even if we fix this language problem by making locks part of the method signature, this rule makes it extremely difficult to implement symmetric accesses, where interactions can originate from either end. And no such fix gets around the extreme difficulty of reasoning about mutual exclusion locks. If programmers cannot understand their code, the code will not be reliable.

We might conclude that the problem lies in how Java realizes threads. Perhaps the synchronized keyword is not the best pruning tool. Indeed, version 5.0 of Java, introduced in 2005, added several other mechanisms for synchronizing threads. These mechanisms do enrich the programmer's toolkit for pruning nondeterminacy. But using mechanisms such as semaphores still requires considerable sophistication, and the result likely will still be incomprehensible programs with subtle lurking bugs.

> **Relatively rigorous software engineering identified and fixed many concurrency bugs, but a problem as serious as a deadlock went undetected.**

Design patterns

Software engineering process improvements alone will not do the job. Another helpful approach uses vetted design patterns for concurrent computation.[8,9] Indeed, these are an enormous help when the problem being solved matches one of the patterns.

However, this approach presents two difficulties. First, implementation of the patterns, even with careful instructions, is still subtle and tricky. Programmers will make errors, and there are no scalable techniques for automatically checking compliance of implementations to patterns. Second, combining the patterns can be difficult. Because their properties are not typically composable, nontrivial programs that require using more than one pattern are unlikely to be understandable.

Databases are an example of a common use of patterns in concurrent computation, particularly with the notion of transactions. Transactions support speculative unsynchronized computation on a copy of the data followed by a commit or abort. A commit occurs when it can be shown that no conflicts have occurred.

Transactions can be supported on distributed hardware, as is common for databases; in software on shared-memory machines; or, most interestingly, in hardware on shared-memory machines. In the latter case, the technique meshes well with the cache consistency protocols required anyway on these machines.

Although transactions eliminate unintended deadlocks, despite recent extensions for composability,[10] they remain a highly nondeterministic interaction mechanism. They are well-suited to intrinsically nondeterminate situations, where for example multiple actors compete nondeterministically for resources. But they are poorly suited for building determinate concurrent interactions.

MapReduce[11] is a particularly interesting use of patterns inspired by the higher-order functions found in Lisp and other functional languages. Google has used this pattern

for large-scale distributed processing of huge data sets. Whereas most patterns provide fine-grained shared-data structures with synchronization, MapReduce provides a framework for the construction of large distributed programs. The pattern's parameters are pieces of functionality represented as code rather than as pieces of data.

Experts can encapsulate patterns into libraries as with the concurrent data structures in Java 5.0 and STAPL in C++. Although this greatly improves the reliability of implementations, constraining all concurrent interactions to occur via these libraries requires some programmer discipline. Folding the capabilities of these libraries into languages in which syntax and semantics enforce these constraints could eventually lead to more easily constructed concurrent programs.

Higher-order patterns such as MapReduce offer some particularly interesting challenges and opportunities for language designers. These patterns function at the level of coordination languages rather than more traditional programming languages. New coordination languages compatible with established programming languages, such as Java and C++, are more likely to gain acceptance than new programming languages that replace established ones.

A common compromise extends established programming languages with annotations or a few selected keywords to support concurrent computation. This compromise admits the reuse of significant portions of legacy code when concurrency is not an issue, but requires rewriting to expose concurrency. For example, Split-C[12] and Cilk[13]—both C-like languages supporting multithreading—follow this strategy.

A related approach combines language extensions with constraints that limit the expressiveness of established languages to get more consistent and predictable behavior. For example, Guava[14] constrains Java so that it cannot access unsynchronized objects from multiple threads. It further makes explicit the distinction between locks that ensure the integrity of read data and locks that enable safe modification of the data.

These language changes prune away considerable nondeterminacy without sacrificing much performance, but they still have deadlock risk.

Other techniques

Promises, also called futures, provide another approach that puts more emphasis on the avoidance of deadlock, as seen in the E programming language (www.erights.org/). Here, instead of blocking access to shared data, programs proceed with a proxy of the data they expect to get eventually, using the proxy as if it were the data itself.

> **Promises, also called futures, provide another approach that puts more emphasis on the avoidance of deadlock, as seen in the E programming language.**

Yet another approach leaves the programming languages and mechanisms for expressing concurrency unchanged, instead introducing formal program analysis to identify potential concurrency bugs in multithreaded programs. This is done, for example, in the Intel thread checker and can help considerably by revealing program behaviors difficult for a human to spot. Similarly, less formal techniques, such as performance debuggers, can also help, making it easier for programmers to sort through the vast nondeterminacy of program behaviors.

Although promising, applying both the formal and informal techniques still requires considerable expertise, and the techniques also suffer from scalability limitations.

All these techniques prune away some of the threads' nondeterminacy. However, they all still result in nondeterministic programs. For applications with intrinsic nondeterminacy, such as servers, concurrent database accesses, or competition for resources, this is appropriate. But achieving deterministic aims through nondeterministic means remains difficult.

Achieving deterministic concurrent computation requires approaching the problem differently. Instead of starting with a highly nondeterministic mechanism like threads and relying on the programmer to prune that nondeterminacy, we should start with deterministic, composable mechanisms and introduce nondeterminism only where needed.

ALTERNATIVES TO THREADS

Consider again the simple observer pattern shown in Figures 1 and 2, which is not so easy to implement using threads.

Rendezvous director

Now, consider Figure 3, which shows the observer pattern implemented in Ptolemy II's *Rendezvous domain*.[6] The box at the upper left, labeled *Rendezvous director*, provides an annotation specifying that this diagram represents a CSP-like concurrent program, in which each component, represented by an icon, is a process, and communication is by rendezvous. The system specifies the processes themselves using ordinary Java code, so this framework is properly viewed as a coordination language, which happens to have a visual syntax.

Reo[15] inspired the Ptolemy II implementation of this rendezvous domain, which includes a *Merge* block that specifies a conditional rendezvous. In the diagram, the block specifies that either of the two Value Producers can rendezvous with both the Value Consumer and Observer. That is, two possible three-way rendezvous interactions can occur, repeatedly and in nondeterministic order.

Once the icons' meanings become clear, the diagram expresses the observer pattern. Everything about the program is deterministic except the explicitly nondeterministic interaction specified by the Merge block. Were that block absent, the program would specify deterministic interactions between deterministic processes. Deadlock is provably absent—in this case, the lack of cycles in the diagram ensures no deadlock. The multiway rendezvous ensures that the Value consumer and Observer see new values in the same order. The observer pattern becomes trivial, as it should be.

PN director

Now that the trivial programming problem is truly trivial, we can start to consider interesting elaborations. We can replace the Rendezvous director in Figure 3 with a "PN director" that realizes the Kahn process networks (PN) model of concurrency.[16] In this model, each icon again represents a process, but instead of rendezvous-based interactions, the processes communicate via message passing with conceptually unbounded FIFO queues and blocking reads.

In the original PN model, the blocking reads ensure that every network defines a deterministic computation. In this case, the Merge block explicitly merges streams nondeterministically. Augmenting the PN model with such explicit nondeterminism is common for embedded software applications.[17]

The PN implementation has all the cited advantages of the implementation in Figure 3, with the added property that the Observer need not keep up with the Value consumer. Notifications can be queued for later processing. In a thread-based implementation, we will unlikely ever get to the point of asking such questions because the programmer effort to get any form of the observer pattern correct is so excessive.

SR director

A third implementation would elaborate on the nature of the nondeterminism that the nondeterministic merge represents. The implementation could use the principles of synchronous languages[18] to ensure fairness. In Ptolemy II, the same model can be implemented with an SR (synchronous/ reactive) director, which implements a synchronous model related to Esterel, SIGNAL, and Lustre. The last of these has been used successfully to design highly concurrent, safety-critical software for aircraft-control applications. Using threads for such applications would not be wise.

DE director

A fourth implementation would focus on the timing of nondeterministic events. In Ptolemy II, a similar model using the DE (discrete events) director would provide a timed specification with rigorous semantics related to that of hardware description languages such as VHDL
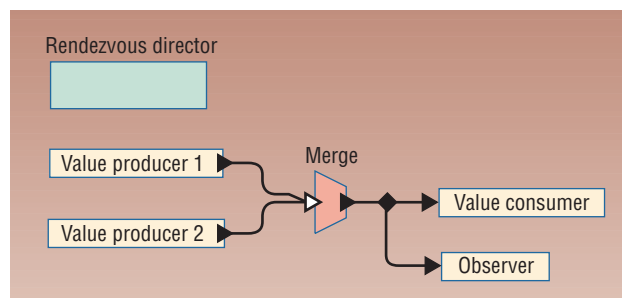
Figure 3. Observer pattern implemented in a rendezvous-based coordination language with a visual syntax. Two possible three-way rendezvous interactions can occur, repeatedly and in nondeterministic order.

and Verilog and to network modeling frameworks such as Opnet Modeler.

Judicious nondeterminism

In all four cases—rendezvous, PN, SR, and DE—we started with an interaction mechanism that was deterministic in how it performed the computation, although in the first three cases it was not deterministic in the sense of timing. These designs judiciously introduce nondeterminism exactly where needed. This style of design differs from the threads style, which starts with a wildly nondeterministic interaction mechanism and attempts to prune away undesired nondeterminism.

The implementation shown in Figure 3 and the PN version both use Java threads. However, the programmer's model does not use threads. Compared to all the techniques described in the previous sections, this is closest to MapReduce, which has a similar flavor of streaming data through computations. But unlike MapReduce, it receives support from a rigorous coordination language sufficiently expressive to describe a wide range of interactions. Four distinct coordination languages are mentioned here, with rendezvous, PN, SR, and DE semantics.

This established style of concurrency, in which data flows through components, has been called "actor-oriented."[19] These architectures can take many forms. Unix pipes resemble PN, although they are more limited in that they do not support cyclic graphs. Message passing packages like MPI and OpenMP include facilities for implementing rendezvous and PN, but in a less structured context that emphasizes expressiveness rather than determinacy. A naive user of such packages can easily encounter unexpected nondeterminacy.

Languages such as Erlang make message-passing concurrency an integral part of a general-purpose language. Languages such as Ada make rendezvous an integral part. Functional languages and single-assignment languages also emphasize deterministic computations, but they are less explicitly concurrent, so controlling and exploiting concurrency can be more challenging. Data-
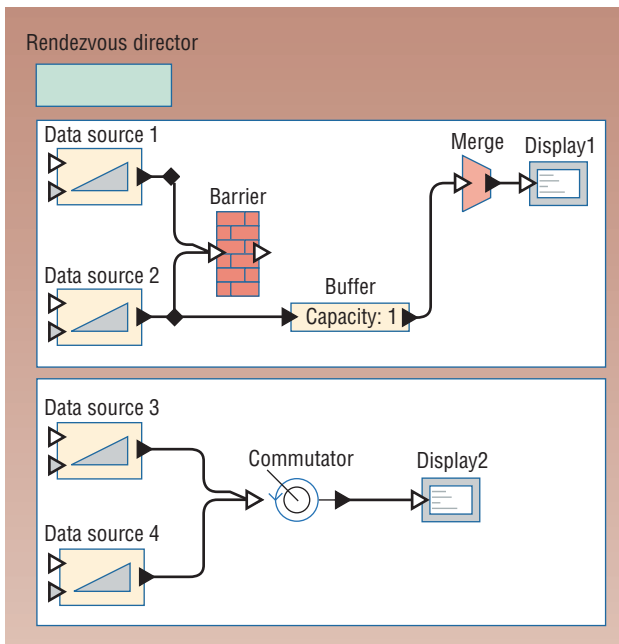
Figure 4. Two ways to accomplish deterministic interleaving using rendezvous. The upper model uses nondeterministic mechanisms to accomplish deterministic aims. In contrast, in the lower model, well-chosen language primitives enable simple, direct, deterministic expression of deterministic aims.

parallel languages also emphasize determinate interactions, but require low-level rewrites of software.

All these approaches offer pieces of the solution. But it seems unlikely that any one will become mainstream.

CHALLENGES AND OPPORTUNITIES

Threads continue to dominate the concurrent programming landscape despite the existence of alternatives. Many obstacles prevent these alternatives from taking root, probably the most important being that the very notion of programming, and the core abstractions of computation, are deeply rooted in the sequential paradigm to which most widely used programming languages adhere. Syntactically, threads provide either a minor extension to these languages, as in Java, or just an external library. Semantically, they thoroughly disrupt the languages' essential determinism.

Regrettably, programmers seem more governed by syntax than semantics. The alternatives to threads that have taken root, like MPI and OpenMP, share this same key feature. They make no syntactic change to languages. Alternatives that replace these languages with entirely new syntax, such as Erlang or Ada, have not taken root, and probably will not. Even languages with minor syntactic modifications to established languages, like Split-C or Cilk, remain esoteric.

The message is clear. We should not replace established languages. We should instead build on them. However, building on them using only libraries is unsat-

isfactory. Libraries offer little structure, no pattern enforcement, and few composable properties.

Coordination languages

The right answer can be found in coordination languages, also called composition languages, which introduce new syntax. That syntax, however, serves purposes orthogonal to those of established programming languages.

Whereas a general-purpose concurrent language like Erlang or Ada must 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.

The program shown in Figure 3 uses a visual syntax to specify actor-oriented coordination. Although here a visual syntax serves only pedagogical purposes, conceivably such visual syntaxes eventually will be made scalable and effective, as certain parts of UML have been for object-oriented programming. If not, we can easily envision scalable textual syntaxes that specify the same structure.

Coordination languages themselves have been around for a long time.[20] They too have failed to take root. One reason for this is 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. Adherents to the language view as traitors any of their colleagues seduced by another language.

A key development, however, has broken the ice. UML—properly viewed as a family of languages, each with a visual syntax—is routinely combined with C++ and Java. Programmers have begun accepting the use of more than one language, especially when the disjoint languages provide complementary features. The program in Figure 3 follows this spirit in that it diagrammatically specifies a large-grained structure quite orthogonal to fine-grained computation.

Concurrency models with stronger determinism than threads, such as Kahn process networks, CSP, and dataflow, have also been available for some time. Some have led to programming languages, such as Occam, and some have led to domain-specific frameworks such as YAPI.[17] Most, however, have principally been used to build elaborate process calculi, and they have not had much effect on mainstream programming. I believe this can change if we use these concurrency models to define coordination languages rather than replacement ones.

Coordination language design

Full of pitfalls, designing good coordination languages is no easier than designing good general-purpose languages. For example, programmers can easily be trapped by the false elegance of a spare set of primitives. In gen-

eral-purpose languages, we know that seven primitives are sufficient, as in Turing machines, but no one builds a serious programming language on these.

Figure 4 shows two implementations of a simple concurrent computation. In the upper program, an adaptation of an example from Farhad Arbab's work,[15] the system deterministically interleaves successive outputs from Data source 1 and 2, which appear in alternating order at the Display block. This is a quite complex, even puzzling, way to provide this rather simple functionality.

In contrast, Figure 4's lower program is easily understood. The Commutator block performs a rendezvous with each of its inputs in top-to-bottom order and thus accomplishes the same interleaving. Judicious choice of language primitives enables simple, direct, and deterministic expressions of deterministic aims. The upper model uses nondeterministic mechanisms, albeit more expressive ones, to accomplish deterministic aims, making it much more obtuse.

Coordination languages must develop scalability and modularity features analogous to those in established languages. Ptolemy II, for example, provides a sophisticated, modern type system at the coordination-language level. Moreover, it offers a preliminary form of inheritance and polymorphism adapted from object-oriented techniques.[19] A huge opportunity exists in adapting the concept of higher-order functions to coordination languages, which would enable constructs like MapReduce at the coordination-language level.

A more challenging, long-term opportunity would adapt the theory of computation to provide better foundations for concurrent computations. Although researchers have made considerable progress in this direction, much more must be done. In addition to the sequential computation modeled as functions mapping bit sequences into bit sequences, a corresponding concurrent model[21] that, instead of a function

$$f : B^{**} \rightarrow B^{**}$$

(see the "Nondeterminism and Threads" sidebar) gives concurrent computation as a function

$$f : (T \rightarrow B^{**}) \rightarrow (T \rightarrow B^{**})$$

with T a partially or totally ordered set of tags, where the ordering can represent time, causality, or more abstract dependency relations. A computation viewed in this way maps an evolving bit pattern into an evolving bit pattern. This basic formulation has been shown to be adaptable to many concurrent computation models.

A chieving concurrency in software is difficult. However, much of this difficulty arises from the abstractions for concurrency we have chosen.

Threads provide the dominant method in use today for general-purpose computing. But nontrivial multithreaded programs are incomprehensible to humans. Design patterns, better granularity of atomicity, improved languages, and formal methods can improve the programming model. These techniques, however, merely chip away at the unnecessarily enormous nondeterminism of the threading model, which remains intrinsically intractable.

If we expect concurrent programming to become mainstream, and if we demand reliability and predictability from programs, we must discard threads as a programming model. We can construct concurrent programming models that are much more predictable and understandable than threads based on a simple principle: Deterministic ends should be accomplished with deterministic means. Nondeterminism should be judiciously and carefully introduced where needed, and it should be explicit in programs. This principle seems obvious, yet threads do not accomplish it. They must be relegated to the engine room of computing, to be suffered only by expert technology providers. ■

References

1. H. Sutter and J. Larus, "Software and the Concurrency Revolution," *ACM Queue*, vol. 3, no. 7, 2005, pp. 54-62.
2. M. Creeger, "Multicore CPUs for the Masses," *ACM Queue*, vol. 3, no. 7, 2005, pp. 63-64.
3. U. Banerjee et al., "Automatic Program Parallelization," *Proc. IEEE*, vol. 81, no. 2, 1993, pp. 211-243.
4. W. Pugh, "Fixing the Java Memory Model," *Proc. Conf. Java Grande*, ACM Press, 1999, pp. 89-98.
5. E. Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.
6. J. Eker et al., "Taming Heterogeneity—The Ptolemy Approach," *Proc. IEEE*, vol. 91, no. 2, 2003, pp. 127-144.
7. H.J. Reekie et al., *Software Practice in the Ptolemy Project*, tech. report series GSRC-TR-1999-01, Gigascale Semiconductor Research Center, Univ. of California, Berkeley, Apr. 1999.
8. D. Lea, *Concurrent Programming in Java: Design Principles and Patterns*, Addison-Wesley, 1997.
9. D.C. Schmidt et al., *Pattern-Oriented Software Architecture—Patterns for Concurrent and Networked Objects*, Wiley, 2000.
10. T. Harris et al., "Composable Memory Transactions," *Proc. ACM Symp. Principles and Practice of Parallel Programming* (PPoPP), ACM Press, 2005, pp. 48-60.

11. J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters, *Proc. 6th Symp. Operating System Design and Implementation* (OSDI), Usenix Assoc., 2004, pp. 137-150.

12. D.E. Culler et al., "Parallel Programming in Split-C," *ACM/IEEE Conf. Supercomputing*, ACM Press, 1993, pp. 262-273.

13. R.D. Blumofe et al., "Cilk: An Efficient Multithreaded Runtime System," *Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPoPP)*, ACM Press, 1995, pp. 207-216.

14. D.F. Bacon, R.E. Strom, and A. Tarafdar, "Guava: A Dialect of Java without Data Races," *ACM SIGPLAN Notices*, vol. 35, 2000, pp. 382-400.

15. F. Arbab, "A Behavioral Model for Composition of Software Components," *L'Object*, to appear 2006.

16. G. Kahn and D.B. MacQueen, "Coroutines and Networks of Parallel Processes," *Information Processing*, B. Gilchrist, ed., North-Holland Publishing, 1977, pp. 993-998.

17. E.A. de Kock et al., "YAPI: Application Modeling for Signal Processing Systems," *Proc. 37th Design Automation Conf.* (DAC 00), ACM Press, 2000, pp. 402-405.

18. A. Benveniste and G. Berry, "The Synchronous Approach to Reactive and Real-Time Systems," *Proc. IEEE*, vol. 79, no. 9, pp. 1270-1282.

19. E.A. Lee and S. Neuendorffer, "Classes and Subclasses in Actor-Oriented Design," *Proc. ACM/IEEE Conf. Formal Methods and Models for Codesign* (MEMOCODE), 2004; http://ptolemy.eecs.berkeley.edu/publications/papers/04/Classes/.

20. G. Papadopoulos and F. Arbab, "Coordination Models and Languages," *Advances in Computers—The Engineering of Large Systems*, vol. 46, M. Zelkowitz, ed., Academic Press, 1998, pp. 329-400.

21. E.A. Lee and A. Sangiovanni-Vincentelli, "A Framework for Comparing Models of Computation," *IEEE Trans. Computer-Aided Design of Circuits and Systems*, vol. 17, no. 12, 1998, pp. 1217-1229.

Edward A. Lee is a professor, chair of the Electrical Engineering Division, and associate chair of the Electrical Engineering and Computer Science Department at the University of California, Berkeley. His research interests include embedded software, concurrency, real-time software, signal processing, and communications. Lee received a PhD in electrical engineering and computer sciences from the University of California, Berkeley. He is a Fellow of the IEEE. Contact him at eal@eecs.berkeley.edu.