Leveraging the full power of multicore processors demands new tools and new thinking from the software industry.

Software and the Concurrency Revolution

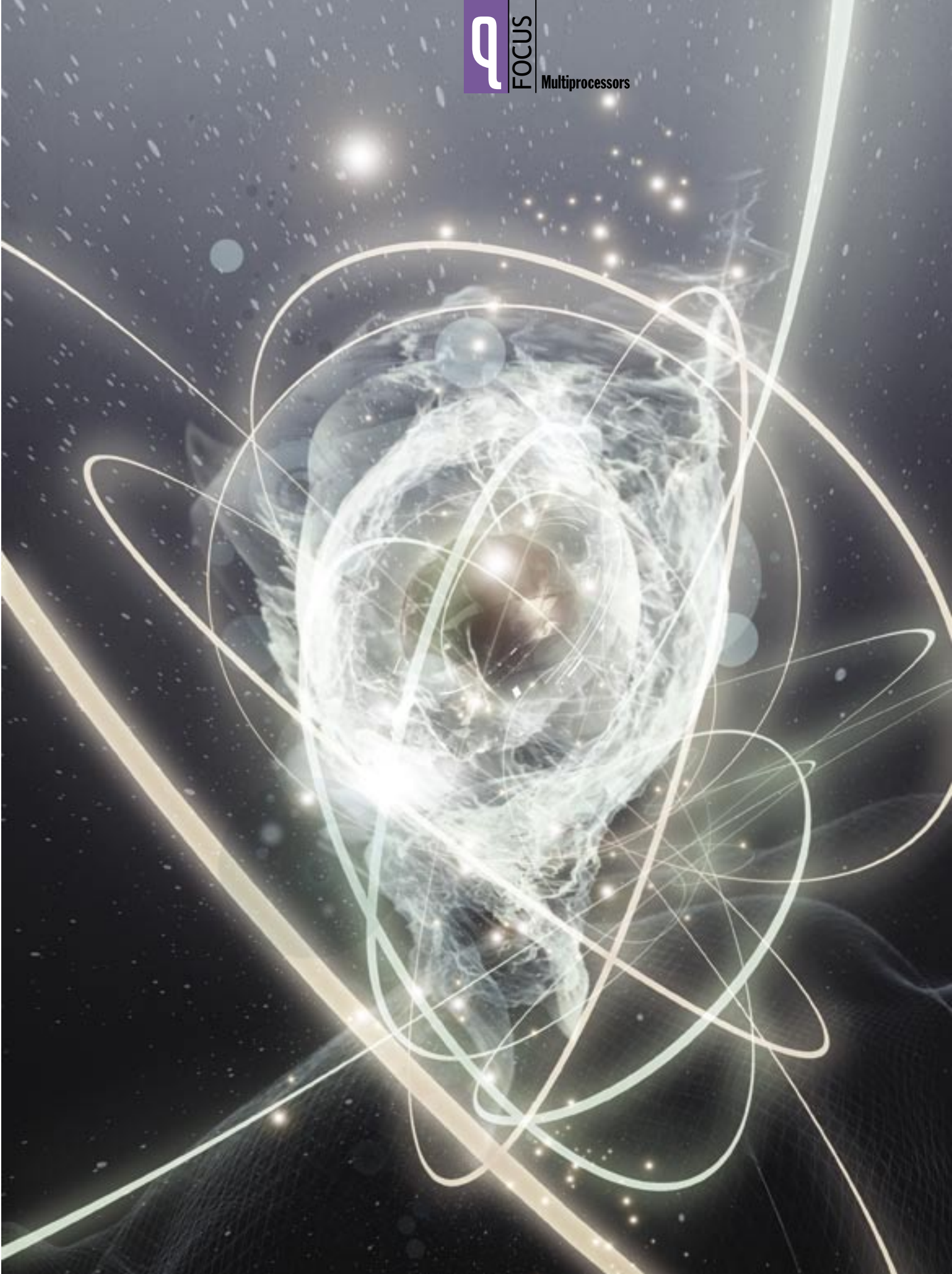
Concurrency has long been touted as the "next big thing" and "the way of the future," but for the past 30 years, mainstream software development has been able to ignore it. Our parallel future has finally arrived: new machines will be parallel machines, and this will require major changes in the way we develop software.

The introductory article in this issue ("The Future of Microprocessors" by Kunle Olukotun and Lance Hammond) describes the hardware imperatives behind this shift in computer architecture from uniprocessors to multicore processors, also known as CMPs (chip multiprocessors). (For related analysis, see "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software."[1])

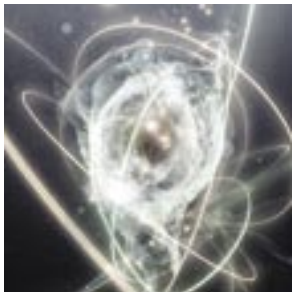
In this article we focus on the implications of concurrency for software and its consequences for both programming languages and programmers.

The hardware changes that Olukotun and Hammond describe represent a fundamental shift in computing. For the past three decades, improvements in semiconductor fabrication and processor implementation produced steady increases in the speed at which computers executed existing sequential programs. The architectural changes in multicore processors benefit only concurrent applications and therefore have little value for most existing mainstream software. For the foreseeable future, today's desktop applications will

HERB SUTTER AND JAMES LARUS, MICROSOFT



Software and the Concurrency Revolution



not run much faster than they do now. In fact, they may run slightly slower on newer chips, as individual cores become simpler and run at lower clock speeds to reduce power consumption on dense multicore processors.

That brings us to a fundamental turning point in software development, at least for mainstream software. Computers will continue to become more and more capable, but programs can no longer simply ride the hardware wave of increasing performance unless they are highly concurrent.

Although multicore performance is the forcing function, we have other reasons to want concurrency: notably, to improve responsiveness by performing work asynchronously instead of synchronously. For example, today's applications must move work off the GUI thread so it can redraw the screen while a computation runs in the background.

But concurrency is hard. Not only are today's languages and tools inadequate to transform applications into parallel programs, but also it is difficult to find parallelism in mainstream applications, and—worst of all—concurrency requires programmers to think in a way humans find difficult.

Nevertheless, multicore machines are the future, and we must figure out how to program them. The rest of this article delves into some of the reasons why it is hard, and some possible directions for solutions.

CONSEQUENCES: A NEW ERA IN SOFTWARE

Today's concurrent programming languages and tools are at a level comparable to sequential programming at the beginning of the structured programming era. Semaphores and coroutines are the assembler of concurrency, and locks and threads are the slightly higher-level structured constructs of concurrency. What we need is OO for concurrency—higher-level abstractions that help build concurrent programs, just as object-oriented abstractions help build large componentized programs.

For several reasons, the concurrency revolution is likely to be more disruptive than the OO revolution. First, concurrency will be integral to higher performance. Languages such as C ignored OO and remained usable for many programs. If concurrency becomes the sole path to higher-performance hardware, commercial and systems programming languages will be valued on their support for concurrent programming. Existing languages, such as C, will gain concurrent features beyond simple models such as pthreads. Languages that fail to support concurrent programming will gradually die away and remain useful only when modern hardware is unimportant.

The second reason that concurrency will be more disruptive than OO is that, although sequential programming is hard, concurrent programming is demonstrably more difficult. For example, context-sensitive analysis of sequential programs is a fundamental technique for taking calling contexts into account when analyzing a program. Concurrent programs also require synchronization analysis, but simultaneously performing both analyses is provably undecidable.[2]

Finally, 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 simple collections of partially ordered operations.

DIFFERENCES BETWEEN CLIENT AND SERVER APPLICATIONS

Concurrency is a challenging issue for client-side applications. For many server-based programs, however, concurrency is a "solved problem," in that we routinely architect concurrent solutions that work well, although programming them and ensuring they scale can still require a huge effort. These applications typically have an abundance of parallelism, as they simultaneously handle many independent request streams. For example, a Web server or Web site independently executes thousands of copies of the same code on mostly nonoverlapping data.

In addition, these executions are well isolated and share state via an abstract data store, such as a database that supports highly concurrent access to structured data. The net effect is that code that shares data through a database can keep its "peaceful easy feeling"—the illusion of living in a tidy, single-threaded universe.

The world of client applications is not nearly as well structured and regular. A typical client application executes a relatively small computation on behalf of a single user, so concurrency is found by dividing a computation into finer pieces. These pieces, say the user interface and program's computation, interact and share data in myriad ways. What makes this type of program difficult

to execute concurrently are nonhomogeneous code; fine-grained, complicated interactions; and pointer-based data structures.

PROGRAMMING MODELS

Today, you can express parallelism in a number of different ways, each applicable to only a subset of programs. In many cases, it is difficult, without careful design and analysis, to know in advance which model is appropriate for a particular problem, and it is always tricky to combine several models when a given application does not fit cleanly into a single paradigm.

These parallel programming models differ significantly in two dimensions: the granularity of the parallel operations and the degree of coupling between these tasks. Different points in this space favor different programming models, so let's examine these axes in turn.

Operations executed in parallel can range from single instructions, such as addition or multiplication, to complex programs that take hours or days to run. Obviously, for small operations, the overhead costs of the parallel infrastructure are significant; for example, parallel instruction execution generally requires hardware support. Multicore processors reduce communication and synchronization costs, as compared with conventional multiprocessors, which can reduce the overhead burden on smaller pieces of code. Still, in general, the finer grained the task, the more attention must be paid to the cost of spawning it as a separate task and providing its communication and synchronization.

The other dimension is the degree of coupling in the communication and synchronization between the operations. The ideal is none: operations run entirely independently and produce distinct outputs. In this case, the operations can run in any order, incur no synchronization or communications costs, and are easily programmed without the possibility of data races. This state of affairs is rare, as most concurrent programs share data among their operations. The complexity of ensuring correct and efficient operation increases as the operations become more diverse. The easiest case is executing the same code for each operation. This type of sharing is often regular and can be understood by analyzing only a single task. More challenging is irregular parallelism, in which the operations are distinct and the sharing patterns are more difficult to comprehend.

INDEPENDENT PARALLELISM

Perhaps the simplest and best-behaved model is independent parallelism (sometimes called "embarrassingly parallel tasks"), in which one or more operations are applied independently to each item in a data collection.

Fine-grained data parallelism relies on the independence of the operations executed concurrently. They should not share input data or results and should be executable without coordination. For example:

```
double A[100][100];
...
A = A * 2;
```

multiplies each element of a 100x100 array by 2 and stores the result in the same array location. Each of the 10,000 multiplications proceeds independently and without coordination with its peers. This is probably more concurrency than necessary for most computers, and its granularity is very fine, so a more practical approach would partition the matrix into $n \times m$ blocks and execute the operations on the blocks concurrently.

At the other end of the granularity axis, some applications, such as search engines, share only a large read-only database, so concurrently processing queries requires no coordination. Similarly, large simulations, which require many runs to explore a large space of input parameters, are another embarrassingly parallel application.

REGULAR PARALLELISM

The next step beyond independent parallelism is to apply the same operation to a collection of data when the computations are mutually dependent. An operation on one piece of data is dependent on another operation if there is communication or synchronization between the two operations.

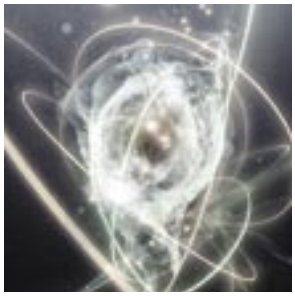
For example, consider a stencil computation that replaces each point in an array, the average of its four nearest neighbors:

```
A[i, j] = (A[i-1, j] + A[i, j-1] + A[i+1, j] + A[i, j+1]) / 4;
```

This computation requires careful coordination to ensure that an array location is read by its neighbors before being replaced by its average. If space is no concern, then the averages can be computed into a new array. In general, other more structured computation strategies, such as traversing the array in a diagonal wavefront, will produce the same result, with better cache locality and lower memory consumption.

Regular parallel programs may require synchronization or carefully orchestrated execution strategies to produce the correct results, but unlike general parallelism, the

Software and the Concurrency Revolution



code behind the operations can be analyzed to determine how to execute them concurrently and what data they share. This advantage is sometimes hypothetical, since program analysis is an imprecise discipline, and sufficiently complex programs are impossible for compilers to understand and restructure.

At the other end of the granularity axis, computations on a Web site are typically independent except for accesses to a common database. The computations run in parallel without a significant amount of coordination beyond the database transactions. This ensures that concurrent access to the same data is consistently resolved.

UNSTRUCTURED PARALLELISM

The most general, and least disciplined, form of parallelism is when the concurrent computations differ, so that their data accesses are not predictable and need to be coordinated through explicit synchronization. This is the form of parallelism most common in programs written using threads and explicit synchronization, in which each thread has a distinct role in the program. In general, it is difficult to say anything specific about this form of parallelism, except that conflicting data accesses in two threads need explicit synchronization; otherwise, the program will be nondeterministic.

THE PROBLEM OF SHARED STATE, AND WHY LOCKS AREN'T THE ANSWER

Another challenging aspect of unstructured parallelism is sharing unstructured state. A client application typically manipulates shared memory organized as unpredictably interconnected graphs of objects.

When two tasks try to access the same object, and one could modify its state, if we do nothing to coordinate the tasks, we have a data race. Races are bad, because the concurrent tasks can read and write inconsistent or corrupted values.

There are a rich variety of synchronization devices that can prevent races. The simplest of these is a lock. Each task that wants to access a piece of shared data must acquire the lock for that data, perform its computation, and then release the lock so other operations on the data can proceed. Unfortunately, although locks work, they pose serious problems for modern software development.

A fundamental problem with locks is that they are not composable. You can't take two correct lock-based pieces of code, combine them, and know that the result is still correct. Modern software development relies on the ability to compose libraries into larger programs, and so it is a serious difficulty that we cannot build on lock-based components without examining their implementations.

The composability issue arises primarily from the possibility of deadlock. In its simplest form, deadlock happens when two locks might be acquired by two tasks in opposite order: task T1 takes lock L1, task T2 takes lock L2, and then T1 tries to take L2 while T2 tries to take L1. Both block forever. Because this can happen any time two locks can be taken in opposite order, calling into code you don't control while holding a lock is a recipe for deadlock.

That is exactly what extensible frameworks do, however, as they call virtual functions while holding a lock. Today's best-of-breed commercial application frameworks all do this, including the .NET Frameworks and the Java standard libraries. We have gotten away with it because developers aren't yet writing lots of heavily concurrent programs that do frequent locking. Many complex models attempt to deal with the deadlock problem—with backoff-and-retry protocols, for example—but they require strict discipline by programmers, and some introduce their own problems (e.g., livelock).

Techniques for avoiding deadlock by guaranteeing locks will always be acquired in a safe order do not compose, either. For example, lock leveling and lock hierarchies prevent programs from acquiring locks in conflicting order by requiring that all locks at a given level be acquired at once in a predetermined order, and that while holding locks at one level, you can acquire additional locks only at higher levels. Such techniques work inside a module or framework maintained by a team (although they're underused in practice), but they assume control of an entire code base. That severely restricts their use in extensible frameworks, add-in systems, and other situations that bring together code written by different parties.

A more basic problem with locks is that they rely on programmers to strictly follow conventions. The relationship between a lock and the data that it protects is implicit, and it is preserved only through programmer discipline. A programmer must always remember to take the right lock at the right point before touching shared

data. Conventions governing locks in a program are sometimes written down, but they're almost never stated precisely enough for a tool to check them.

Locks have other more subtle problems. Locking is a global program property, which is difficult to localize to a single procedure, class, or framework. All code that accesses a piece of shared state must know and obey the locking convention, regardless of who wrote the code or where it resides.

Attempts to make synchronization a local property do not work all the time. Consider a popular solution such as Java's synchronized methods. Each of an object's methods can take a lock on the object, so no two threads can directly manipulate the object's state simultaneously. As long as an object's state is accessed only by its methods and programmers remember to add the synchronized declaration, this approach works.

There are at least three major problems with synchronized methods. First, they are not appropriate for types whose methods call virtual functions on other objects (e.g., Java's Vector and .NET's SyncHashTable), because calling into third-party code while holding a lock opens the possibility of deadlock. Second, synchronized methods can perform too much locking, by acquiring and releasing locks on all object instances, even those never shared across threads (typically the majority). Third, synchronized methods can also perform too little locking, by not preserving atomicity when a program calls multiple methods on an object or on different objects. As a simple example of the latter, consider a banking transfer:

```
account1.Credit(amount); account2.Debit(amount)
```

Per-object locking protects each call, but does not prevent another thread from seeing the inconsistent state of the two accounts between the calls. Operations of this type, whose atomicity does not correspond to a method call boundary, require additional, explicit synchronization.

LOCK ALTERNATIVES

For completeness, we note two major alternatives to locks. The first is *lock-free programming.* By relying on a deep knowledge of a processor's memory model, it is possible to create data structures that can be shared without explicit locking. Lock-free programming is difficult and fragile; inventing a new lock-free data-structure implementation is still often a publishable result.

The second alternative is *transactional memory,* which brings the central idea of transactions from databases into programming languages. Programmers write their programs as a series of explicitly atomic blocks, which appear to execute indivisibly, so concurrently executing operations see the shared state strictly before or after an atomic action executes. Although many people view transactional memory as a promising direction, it is still a subject of active research.

WHAT WE NEED IN PROGRAMMING LANGUAGES

We need higher-level language abstractions, including evolutionary extensions to current imperative languages, so that existing applications can incrementally become concurrent. The programming model must make concurrency easy to understand and reason about, not only during initial development but also during maintenance.

EXPLICIT, IMPLICIT, AND AUTOMATIC PARALLELIZATION

Explicit programming models provide abstractions that require programmers to state exactly where concurrency can occur. The major advantage of expressing concurrency explicitly is that it allows programmers to take full advantage of their application domain knowledge and express the full potential concurrency in the application. It has drawbacks, however. It requires new higher-level programming abstractions and a higher level of programmer proficiency in the presence of shared data.

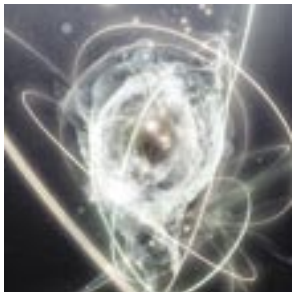
Implicit programming models hide concurrency inside libraries or behind APIs, so that a caller retains a sequential worldview while the library performs the work in parallel. This approach lets naïve programmers safely use concurrency. Its main drawback is that some kinds of concurrency-related performance gains can't be realized this way. Also, it is difficult to design interfaces that do not expose the concurrency in any circumstance—for example, when a program applies the operation to several instances of the same data.

Another widely studied approach is automatic parallelization, where a compiler attempts to find parallelism, typically in programs written in a conventional language such as Fortran. As appealing as it may seem, this approach has not worked well in practice. Accurate program analysis is necessary to understand a program's potential behavior. This analysis is challenging for simple languages such as Fortran, and far more difficult for languages, such as C, that manipulate pointer-based data. Moreover, sequential programs often use sequential algorithms and contain little concurrency.

IMPERATIVE AND FUNCTIONAL LANGUAGES.

Popular commercial programming languages (e.g., Pascal, C, C++, Java, C#) are imperative languages in which a

Software and the Concurrency Revolution



programmer specifies step-by-step changes to variables and data structures. Fine-grained control constructs (e.g., for loops), low-level data manipulations, and shared mutable object instances make programs in these languages difficult to analyze and automatically parallelize.

The common belief is that functional languages, such as Scheme, ML, or Haskell, could eliminate this difficulty because they are naturally suited to concurrency. Programs written in these languages manipulate immutable object instances, which pose no concurrency hazards. Moreover, without side effects, programs seem to have fewer constraints on execution order.

In practice, however, functional languages are not necessarily conducive to concurrency. The parallelism exposed in functional programs is typically at the level of procedure calls, which is impractically fine-grained for conventional parallel processors. Moreover, most functional languages allow some side effects to mutable state, and code that uses these features is difficult to parallelize automatically.

These languages reintroduce mutable state for reasons of expressibility and efficiency. In a purely functional language, aggregate data structures, such as arrays or trees, are updated by producing a copy containing a modified value. This technique is semantically attractive but can be terrible for performance (linear algorithms easily become quadratic). In addition, functional updates do nothing to discourage the writing of a strictly sequential algorithm, in which each operation waits until the previous operation updates the program's state.

The real contribution of functional languages to concurrency comes in the higher-level programming style commonly employed in these languages, in which operations such as map or map-reduce apply computations to all elements of an aggregate data structure. These higher-level operations are rich sources of concurrency. This style of programming, fortunately, is not inherently tied to functional languages, but is valuable in imperative programs.

For example, Google Fellows Jeffrey Dean and Sanjay Ghemawat describe how Google uses Map-Reduce to express large-scale distributed computations.[3] Imperative languages can judiciously add functional style extensions and thereby benefit from those features. This is important because the industry can't just start over. To preserve the huge investment in the world's current software, it is essential to incrementally add support for concurrency, while preserving software developers' expertise and training in imperative languages.

BETTER ABSTRACTIONS

Most of today's languages offer explicit programming at the level of threads and locks. These abstractions are low-level and difficult to reason about systematically. Because these constructs are a poor basis for building abstractions, they encourage multithreaded programming with its problems of arbitrary blocking and reentrancy.

Higher-level abstractions allow programmers to express tasks with inherent concurrency, which a runtime system can then combine and schedule to fit the hardware on the actual machine. This will enable applications that perform better on newer hardware. In addition, for mainstream development, programmers will value the illusion of sequential execution within a task.

Two basic examples of higher-level abstractions are asynchronous calls and futures. An *asynchronous call* is a function or method call that is nonblocking. The caller continues executing and, conceptually, a message is sent to a task, or fork, to execute operation independently. A *future* is a mechanism for returning a result from an asynchronous call; it is a placeholder for the value that has not yet materialized.

Another example of a higher-level abstraction is an *active object*, which conceptually runs on its own thread so that creating 1,000 such objects conceptually creates 1,000 potential threads of execution. An active object behaves as a monitor, in that only one method of the object executes at a given time, but it requires no traditional locking. Rather, method calls from outside an active object are asynchronous messages, marshaled, queued, and pumped by the object. Active objects have many designs, from specialized actor languages to COM single-threaded apartments callable from traditional C code, and many design variables.

Other higher-level abstractions are needed, such as protocols to describe and check asynchronous message exchange. Together they should bring together a consistent programming model that can express typical application concurrency requirements across all of the major granularity levels.

WHAT WE NEED IN TOOLS

Parallel programming, because of its unfamiliarity and intrinsic difficulty, is going to require better programming tools to systematically find defects, help debug programs, find performance bottlenecks, and aid in testing. Without these tools, concurrency will become an impediment that reduces developer and tester productivity and makes concurrent software more expensive and of lower quality.

Concurrency introduces new types of programming errors, beyond those all too familiar in sequential code. Data races (resulting from inadequate synchronization and deadlocks) and livelocks (resulting from improper synchronization) are difficult defects to find and understand, since their behavior is often nondeterministic and difficult to reproduce. Conventional methods of debugging, such as reexecuting a program with a breakpoint set earlier in its execution, do not work well for concurrent programs whose execution paths and behaviors may vary from one execution to the next.

Systematic defect detection tools are extremely valuable in this world. These tools use static program analysis to systematically explore *all* possible executions of a program; thus, they can catch errors that are impossible to reproduce. Although similar techniques, such as model checking, have been used with great success for finding defects in hardware, which is inherently concurrent, software is more difficult. The state space of a typical program is far larger than that of most hardware, so techniques that systematically explore an artifact's states have much more work to do. In both cases, modularity and abstraction are the keys to making the analysis tractable. In hardware model testing, if you can break off the ALU (arithmetic logic unit) and analyze it independently of the register file, your task becomes much more tractable.

That brings us to a second reason why software is more difficulty to analyze: it is far harder to carve off pieces of a program, analyze them in isolation, and then combine the results to see how they work together. Shared state, unspecified interfaces, and undocumented interactions make this task much more challenging for software.

Defect detection tools for concurrent software comprise an active area of research. One promising technique from Microsoft Research called KISS (Keep it Strictly Sequential)[4] transforms a threaded program into a sequential program whose execution behavior includes all possible interleaves of the original threads that involve no more than two context switches. The transformed program can then be analyzed by the large number of existing sequential tools, which then become concurrent defect detection tools for this bounded model.

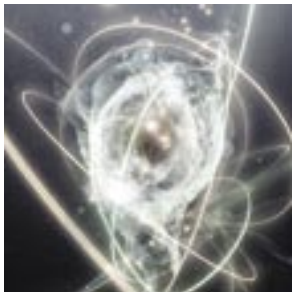
Even with advances such as these, programmers are still going to need good debuggers that let them understand the complex and difficult-to-reproduce interactions in their parallel programs. There are two general techniques for collecting this information. The first is better logging facilities that track which messages were sent to which process or which thread accessed which object, so that a developer can look back and understand a program's partially ordered execution. Developers will also want the ability to follow causality trails across threads (e.g., which messages to one active object, when executed, led to which other messages to other active objects?), replay and reorder messages in queues, step through asynchronous call patterns including callbacks, and otherwise inspect the concurrent execution of their code. The second approach is reverse execution, which permits a programmer to back up in a program's execution history and reexecute some code. Replay debugging is an old idea, but its cost and complexity have been

barriers to adoption. Recently, virtual machine monitors have reduced both factors.[5] In a concurrent world, this technique will likely become a necessity.

Performance debugging and tuning will require new tools in a concurrent world as well. Concurrency introduces new performance bottlenecks, such as lock contention, cache coherence overheads, and lock convoys, which are often difficult to identify with simple profilers. New tools that are more aware of the underlying computer architecture and the concurrent structure of a program will be better able to identify these problems.

Testing, too, must change. Concurrent programs, because of their nondeterministic behaviors, are more difficult to test. Simple code coverage metrics, which track whether a statement or branch has executed, need to be extended to take into account the other code that is executing concurrently, or else testing will provide

Software and the Concurrency Revolution



an unrealistically optimistic picture of how completely a program has been exercised. Moreover, simple stress tests will need to be augmented by more systematic techniques that use model-checking-like techniques to explore systems' state spaces. For example, Verisoft has been very successful in using these techniques to find errors in concurrent telephone switching software.[6] Today, many concurrent applications use length of stress testing to gain confidence that the application is unlikely to contain serious races. In the future, that will increasingly be insufficient, and software developers will need to be able to prove their product's quality through rigorous deterministic testing instead of relying on a probabilistic confidence based on stress tests.

PARALLELISM IS KEY

The concurrency revolution is primarily a software revolution. The difficult problem is not building multicore hardware, but programming it in a way that lets mainstream applications benefit from the continued exponential growth in CPU performance.

The software industry needs to get back into the state where existing applications run faster on new hardware. To do that, we must begin writing concurrent applications containing at least dozens, and preferably hundreds, of separable tasks (not all of which need be active at a given point).

Concurrency also opens the possibility of new, richer computer interfaces and far more robust and functional software. This requires a new burst of imagination to find and exploit new uses for the exponentially increasing potential of new processors.

To enable such applications, programming language designers, system builders, and programming tool creators need to start thinking seriously about parallelism and find techniques better than the low-level tools of threads and explicit synchronization that are today's basic building blocks of parallel programs. We need higher-level parallel constructs that more clearly express a programmer's intent, so that the parallel architecture of a program is more visible, easily understood, and verifiable by tools. Q

REFERENCES

1. Sutter, H. 2005. The free lunch is over: a fundamental turn toward concurrency in software. *Dr. Dobb's Journal* 30 (3); http://www.gotw.ca/publications/concurrency-ddj.htm.
2. Ramalingam, G. 2000. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Transactions on Programming Languages and Systems* 22 (2): 416-430.
3. Dean, J., and Ghemawat, S. 2004. MapReduce: simplified data processing on large clusters. *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation*, San Francisco, CA: 137-150.
4. Qadeer, S., and Wu, D. 2004. KISS: Keep it Simple and Sequential. *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, Washington, DC: 1-13.
5. King, S. T., Dunlap, G. W., and Chen, P. M. 2005. Debugging operating systems with time-traveling virtual machines. *Proceedings of the 2005 Annual Usenix Technical Conference*, Anaheim, CA: 1-15.
6. Chandra, S., Godefroid, P., and Palm, C. 2002. Software model checking in practice: an industrial case study. *Proceedings of the 24th International Conference on Software Engineering*, Orlando, FL: 431-441.

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

HERB SUTTER is a software architect in Microsoft's developer division. He chairs the ISO C++ standards committee, and is the author of four books and more than 200 technical papers and articles, including the widely read "The Free Lunch Is Over" essay on the concurrency revolution. He can be reached at hsutter@microsoft.com.

JAMES LARUS is a senior researcher at Microsoft Research, managing SWIG (Software Improvement Group), which consists of the SPT (software productivity tools), TVM (testing, verification, and measurement), and HIP (human interactions in programming) research groups, and running the Singularity research project. Before joining Microsoft, he was an associate professor at the University of Wisconsin-Madison, where he co-led the Wisconsin Wind Tunnel research project. This DARPA- and NSF-funded project investigated new approaches to building and programming parallel shared-memory computers. Larus received his Ph.D. in computer science from the University of California at Berkeley.