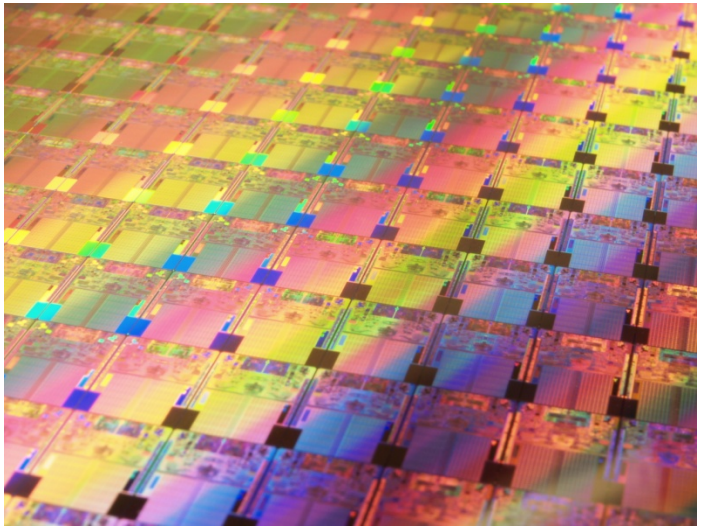
How to Survive the Multicore Software Revolution (or at Least Survive the Hype)

Charles E. Leiserson
Ilya B. Mirman

CILK ARTS



Contents

Preface iv

1. The Emergence of Multicore Software 1

The free lunch is over 1

The multicore software triad 2

2. What Is Parallelism, Anyhow? 3

Amdahl's Law 3

A model for multithreaded execution 4

Work 5

Span 6

Parallelism 6

Amdahl's Law Redux 7

3. Race Conditions: A New Type of Bug (for Most People) 8

The lure of nonlocal variables 8

Race conditions 9

A simple example 10

Using the dag model 10

Atomicity 11

Validating determinacy 12

Coping with race bugs 13

4. The Folly of Do-It-Yourself Multithreading 14

Three desirable properties 14

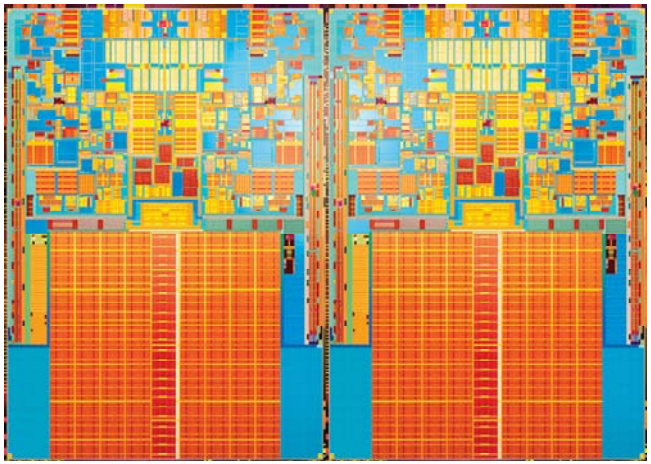
A tiny example 15

A version using native threads 15
The impact on development time 16
The bottom line 17
5. Concurrency Platforms 18
Thread pools 19
Message passing 20
Data-parallel languages 22
Intel's Threading Building Blocks 24
OpenMP 25
Cilk++ 26
6. Twenty Questions to Ask When Going Multicore 29
Application performance 30
Software reliability 30
Development time 31
About the Authors 31
About Cilk Arts 32
Index 33

© 2008 Cilk Arts, Inc. All rights reserved.

Rev. R21.3

Preface



Intel 45nm quad-core processor

An irreversible shift towards multicore x86 processors is underway. Building multicore processors delivers on the promise of Moore's Law, but it creates an enormous problem for developers. Multicore processors are parallel computers, and parallel computers are notoriously difficult to program.

To deliver competitive application performance on these new processors, many applications must be written (or rewritten) as parallel, multithreaded applications. Multithreaded development can be difficult, expensive, time-consuming, and error-prone — and it requires new programming skill sets. Organizations need a solution to meet the multicore software challenge.

To help you survive and prosper during the multicore revolution, we have in this e-Book tried to provide some background and context around the emergence of mainstream multicore processors, identify the key challenges facing software developers, provide an introduction to multithreading concepts, and overview several concurrency platforms available today.

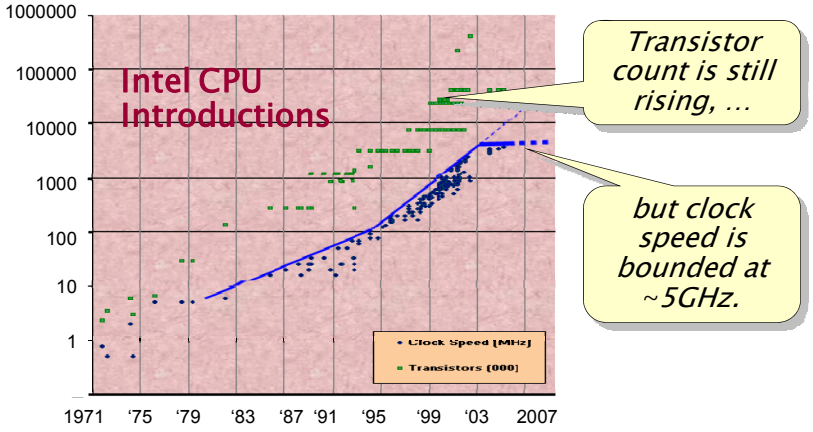
We have tried hard to put together an informative, objective, and vendor-neutral resource. Occasionally, however, we mention the features and benefits of our Cilk++ product. To make those "commercial breaks" clear, we identify them using the icon on the left.

We welcome your feedback on the content! If you have suggestions on ways to improve this e-Book, as well as additional topics to include, we look forward to hearing from you at info [at] cilk.com or on our blog (www.cilk.com/multicore-blog/).

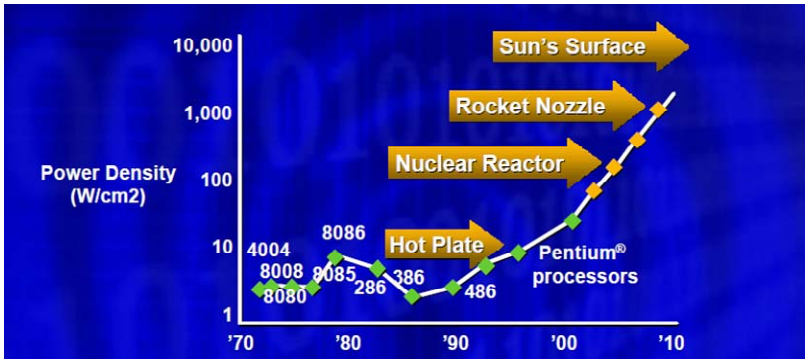
Good Luck in Going Multicore!

The Cilk Artisans

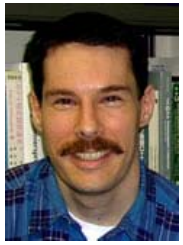
Cilk, Cilk++, and Cilkscreen are trademarks of Cilk Arts, Inc.



Source: Herb Sutter, "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software," *Dr. Dobb's Journal*, 30(3), March 2005.



Source: Patrick Gelsinger, *Intel Developer's Forum*, Intel Corporation, 2004.



Herb Sutter

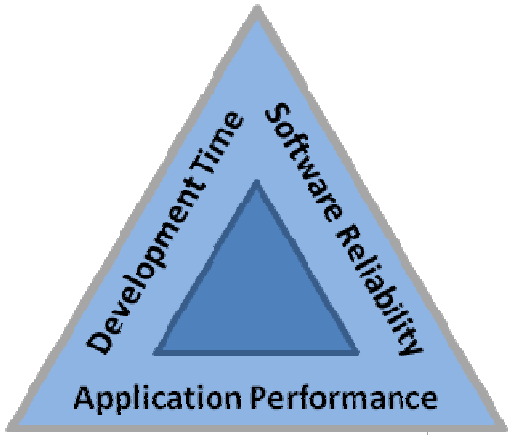
1. The Emergence of Multicore Software

In a 1965 paper, Intel cofounder Gordon Moore observed that transistor density increases exponentially, roughly doubling every 18 months or so. For the last 40 years, ***Moore's Law*** has continued unabated: semiconductor density continues to increase. Moreover, since the mid-1980's, a similar trend has resulted in clock speed increases of about 30% per year. Around 2003, however, clock speed hit a wall due to fundamental physics. Although computing power increases linearly with clock speed, power density rises with the square or cube, depending on the electrical model, and clock frequencies beyond about 5GHz melt chips. The clock-speed wall has forced microprocessor vendors, in their efforts to continue leveraging Moore's Law, to increase performance and reduce power through multiprocessing, producing ***chip multiprocessors*** with multiple ***processing cores*** per chip. Instead of increasing clock rate, vendors are doubling the number of processing cores with each new generation of microprocessor chips, a trend which will continue for the foreseeable future.

The free lunch is over

In the words of Microsoft C++ guru Herb Sutter, for software developers, the "free lunch" of performance — in the form of ever faster clock speeds — is over. Because multicore programming differs so greatly from the serial software technology that has carried the software industry through five decades of Moore's Law, software developers face an unprecedented challenge to their substantial investment in legacy serial codebases. Moreover, every generation of multicore processors widens the "software gap" between hardware potential and the performance that can be delivered by today's applications. This multicore discontinuity has placed three new demands on the software development industry:

1. If software developers wish to continue to deliver competitive levels of application performance, they must identify and adopt multicore software platforms that allow them to exploit the full capabilities of multicore architectures.



The Multicore Software Triad

2. Since multicore enablement requires multithreading expertise, which is in short supply, software vendors must engage in a massive training effort to develop the skills needed to cope with multicore programming.
3. The entire software tool chain, from debugging to software release, must be engineered to allow reliable multicore software to be developed.

The multicore software triad

Developing multicore software poses three key challenges: achieving high application performance, ensuring software reliability, and minimizing development time. We call these three requirements the ***multicore software triad***.

Application performance

Software developers need best-in-class performance, but not only on today's 2- or 4-core machines. They would like to avoid rewriting their code each time the number of cores increases, and they would like to use a single binary with all of their customers — including those who still run on 1- or 2-core machines. That is, they need linear scaling up and down, including low overheads on a single core.

Software reliability

When parallelism is introduced into an application, that application becomes vulnerable to "race conditions." A race condition occurs when concurrent software tasks access a shared-memory location and at least one of the tasks stores a value into the location. Depending on the scheduling of the tasks, the software may behave differently. Coping with race conditions is particularly challenging, as these bugs are nondeterministic, making them difficult to avoid, hard to find during testing, and in some cases, expensive to resolve.

Development time

Developing multithreaded software can be dramatically more complex than developing serial code. To multithread their software, developers are faced with a potentially cataclysmic restructuring of their code bases. The complexity may require organizations to acquire new programming skill sets, forcing retraining or retooling of development

teams. These factors can put enormous pressure on development schedules and introduce risk.

The remainder of this e-book is organized as follows. The next three chapters address salient issues and concepts regarding each of legs of the multicore-software triad in turn. Chapter 5 then overviews several popular concurrency platforms, and Chapter 6 provides 20 questions to ask when going multicore.

2. What Is Parallelism, Anyhow?

Gene M. Amdahl

Before multicore-enabling a legacy codebase, it's prudent to understand something about the application performance that one can expect to achieve through parallelization. Many will be familiar with ***Amdahl's Law***[1], originally proffered by Gene Amdahl in 1967, which offers a somewhat gloomy assessment of how much speedup can be obtained through parallelism. This chapter introduces a simple theoretical model for parallel computing which provides a more general and precise quantification of parallelism that subsumes Amdahl's Law. With this little theory in hand, we can evaluate Amdahl's Law and its implications on multicore application performance.

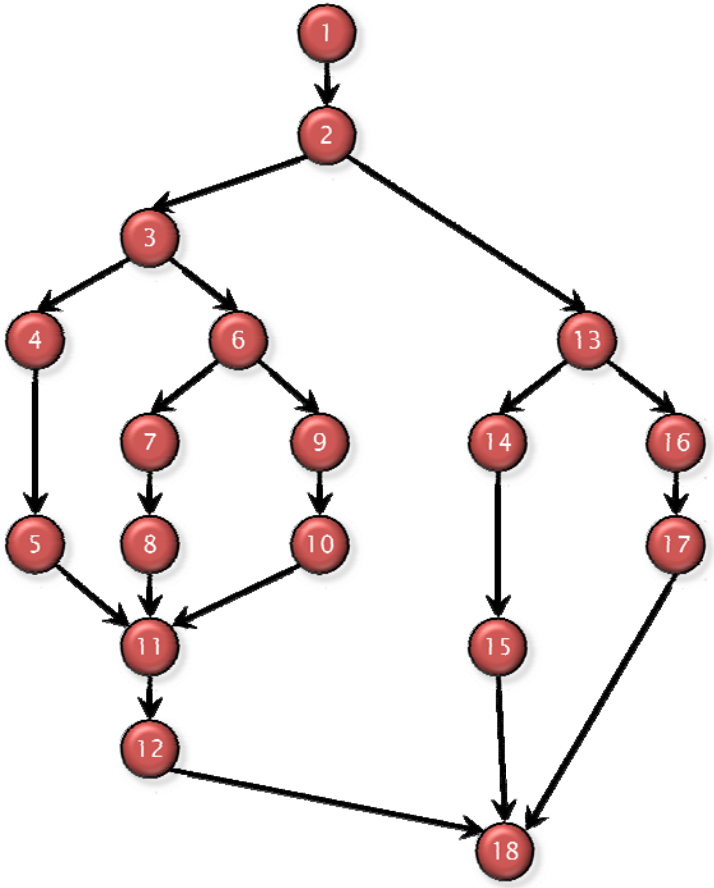
Amdahl's Law

Let's begin by looking at Amdahl's Law to see what it says and what it doesn't say. Amdahl made what amounts to the following observation. Suppose that 50% of a computation can be parallelized and 50% can't. Then, even if the 50% that is parallel were run on an infinite number of processors, the total time is cut at most in half, leaving a speedup of less than 2. In general, if a fraction p of a computation can be run

1. **Amdahl, Gene**. "The validity of the single processor approach to achieving large-scale computing capabilities," *Proceedings of the AFIPS Spring Joint Computer Conference,* April 1967, pp. 483-485.

in parallel and the rest must run serially, Amdahl's Law upper-bounds the speedup by $1/(1-p)$.

This argument was used in the 1970's and 1980's to argue that parallel computing, which was in its infancy back then, was a bad idea — the implication being that most applications have long, inherently serial subcomputations that limit speedup. We now know from numerous examples that there are plenty of applications that can be sped up on parallel computers effectively, but Amdahl's Law doesn't really help in understanding how much speedup you can expect from *your* application. After all, few applications can be decomposed so simply into just a serial part and a parallel part.

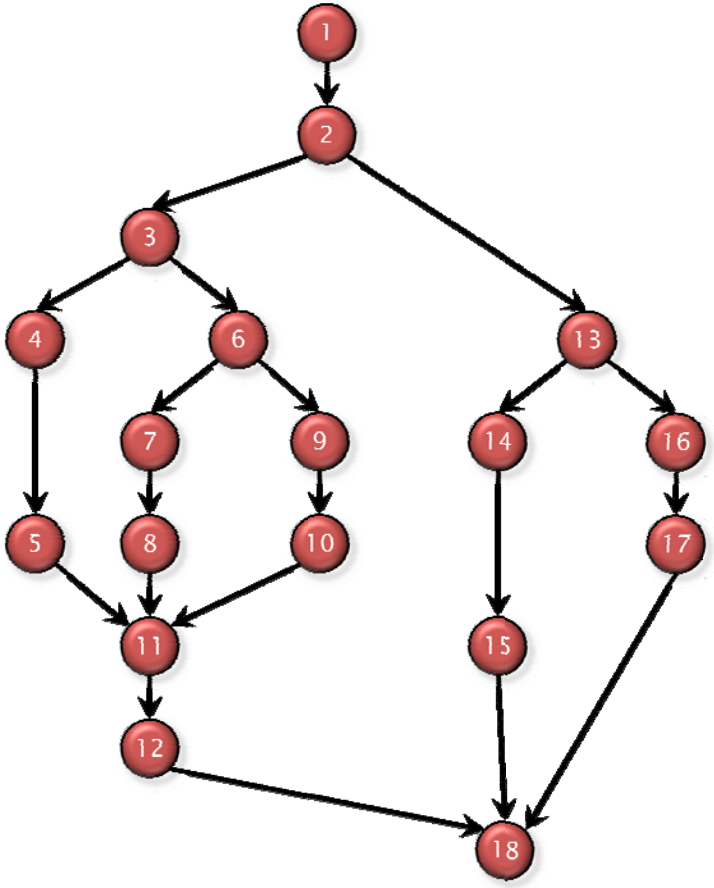


A model for multithreaded execution

As with much of theoretical computer science, we need a model which abstracts away inessential properties and let's focus on what we care about more precisely. The appropriate model for studying parallelism is the ***dag model of multithreading***, which models the series-parallel relationships in terms of a ***dag***, or ***directed acyclic graph***. This simple model views the execution of a multithreaded program as a set of instructions (the vertices of the dag) with graph edges indicating dependences between instructions.

We say that an instruction x ***precedes*** an instruction y, sometimes denoted $x \prec y$, if x must complete before y can begin. In a diagram for the dag, $x \prec y$ means that there is a positive-length path from x to y. If neither $x \prec y$ nor $y \prec x$, we say the instructions are in ***parallel***, denoted $x \parallel y$. The figure to the left illustrates a multithreaded dag. In the figure, we have, for example, $1 \prec 2$, $6 \prec 12$, and $4 \parallel 9$.

Two measures of the dag allow us to define parallelism precisely, as well as to provide some key bounds on performance and speedup.



Work

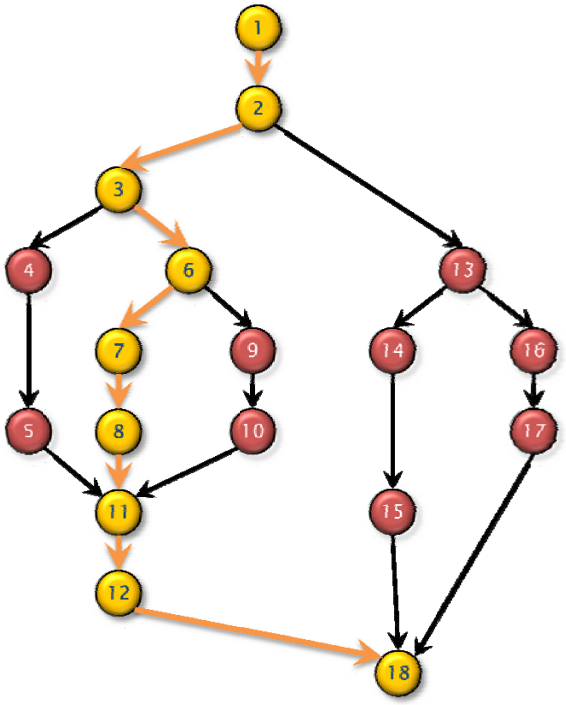
The first important measure is ***work***, which is the total amount of time spent in all the instructions. Assuming for simplicity that it takes unit time to execute an instruction, the work for the example dag is 18. (This simple theoretical model has been extended in the literature to handle effects such as nonunit instruction times, caching, etc., but for now, they simply complicate our understanding.)

Let's adopt a simple notation. Let T_P be the fastest possible execution time of the application on P processors. Since the work corresponds to the execution time on 1 processor, we denote it by T_1. Among the reasons that work is an important measure is because it provides a bound on P-processor execution time:

Work Law: $T_P \geq T_1/P$

This Work Law holds, because in our model, each processor executes at most 1 instruction per unit time, and hence P processors can execute at most P instructions per unit time. Thus, to do all the work on P processors, it must take at least $T1/P$ time.

We can interpret the Work Law in terms of speedup. Using our notation, the ***speedup*** on P processors is just T_1/T_P, which is how much faster the application runs on P processors than on 1 processor. Rewriting the Work Law, we obtain $T_1/T_P \leq P$, which is to say that the speedup on P processors can be at most P. If the application obtains speedup proportional to P, we say that the application exhibits ***linear speedup***. If it obtains speedup exactly P (which is the best we can do in our model), we say that the application exhibits ***perfect linear speedup***. If the application obtains speedup greater than P (which can't happen in our model due to the work bound, but can happen in models that incorporate caching and other processor effects), we say that the application exhibits ***superlinear speedup***.



Span

The second important measure is ***span***, which is the longest path of dependences in the dag. The span of the dag in the figure is 9, which corresponds to the path 1→ 2 → 3→ 6 → 7 → 8 → 11→ 12 → 18. This path is sometimes called the ***critical path*** of the dag, and span is sometimes referred to in the literature as ***critical-path length***. Since the span is the theoretically fastest time the dag could be executed on a computer with an infinite number of processors (assuming no overheads for communication, scheduling, etc.), we denote it by T_∞. Like work, span also provides a bound on P-processor execution time:

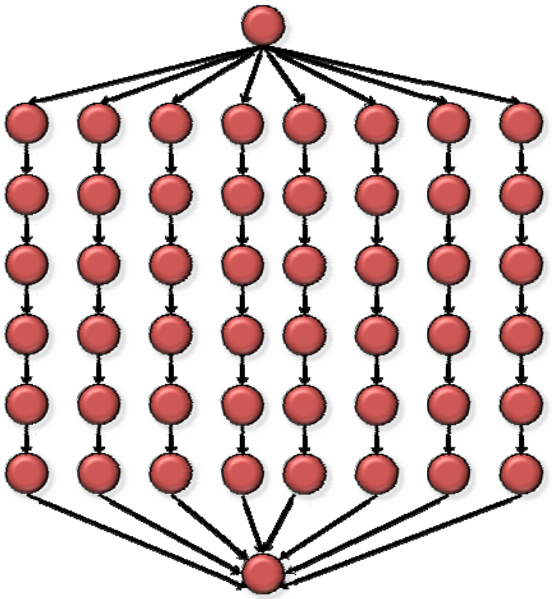
Span Law: $T_P \geq T_\infty$

This Span Law arises for the simple reason that a finite number of processors cannot outperform an infinite number of processors, because the infinite-processor machine could just ignore all but P of its processors and mimic a P-processor machine exactly.

Parallelism

Parallelism is defined as the ratio of work to span, or T_1/T_∞. Why does this definition make sense? There are several ways to understand it:

1. The parallelism T_1/T_∞ is the average amount of work along each step of the critical path.
2. The parallelism T_1/T_∞ is the maximum possible speedup that can be obtained by any number of processors.
3. Perfect linear speedup cannot be obtained for any number of processors greater than the parallelism T_1/T_∞. To see this third point, suppose that $P > T_1/T_\infty$, in which case the Span Law $T_P \geq T_\infty$ implies that the speedup T_1/T_P satisfies $T_1/T_P \leq T_1/T_\infty < P$. Since the speedup is strictly less than P, it cannot be perfect linear speedup. Note also that if $P \gg T_1/T_\infty$, then $T_1/T_P \ll P$ — the more processors you have beyond the parallelism, the less "perfect" the speedup.



Work: $T_1 = 50$

Span: $T_\infty = 8$

Parallelism: $T_1/T_\infty = 6.25$

John Gustafson

Amdahl's Law Redux

Amdahl's Law for the situation where a fraction p of the application is parallel and a fraction 1–p is serial simply amounts to the special case where $T_\infty > (1–p)\ T_1$. In this case, the maximum possible speedup is $T_1/T_\infty < 1/(1–p)$. Amdahl's Law is simple, but the Work and Span Laws are far more powerful.

Twenty years ago, John Gustafson[2] addressed the gloominess of Amdahl's law making the observation that as time marches forward, problem sizes tend to get bigger, and hence parallelism increases. In algebraic terms, if a problem of size n has work $T_1(n)$ and span $T_\infty(n)$, the parallelism is $T_1(n)/T_\infty(n)$, which increases with n, and hence with time, if this ratio is not constant. For example, quicksorting n numbers has parallelism proportional to log n, since the work is O(n log n) and the span is order n (mainly due to the partitioning). That's not much parallelism, compared to n x n matrix multiplication, which has parallelism of almost n^3. Nevertheless, since we're dealing with larger and larger data sets as time goes on due to Moore's Law, the parallelism grows with time, once again, assuming the parallelism isn't a constant.

Another reason Amdahl's Law is gloomier than reality is that people tend to be more interested in response time than throughput. If one has an interactive application for which most operations can be done serially, but one operation takes a long time but is parallelizable, it may be well worth the effort to parallelize the one operation, even though it contributes to only a small part of the total workflow. Moreover, making an operation cheap can change the user's behavior, encouraging a workflow in which this operation occurs more frequently — a positive feedback loop which results in a larger fraction of the workflow that is parallelizable. Amdahl analyzed a static case, ignoring the dynamics of human-centered computing.

---

2. **Gustafson, John L**. "Reevaluating Amdahl's Law," *Communications of the ACM,* May 1988, pp. 532–533.

In closing, we should mention that the theory of work and span has led to an excellent understanding of multithreaded scheduling, at least for those who know the theory. As it turns out, scheduling a multithreaded computation to achieve optimal performance is NP-complete, which in lay terms means that it is computationally intractable. Nevertheless, practical scheduling algorithms exist based on work and span that can schedule any multithreaded computation near optimally. The Cilk++ runtime platform contains such a near-optimal scheduler.

3. Race Conditions: A New Type of Bug (for Most People)

In a widely applauded article published in 1973 and entitled, "Global variable considered harmful," Bill Wulf and Mary Shaw argued, "We claim that the non-local variable is a major contributing factor in programs which are difficult to understand." Thirty-five years later, however, nonlocal variables are still very much in vogue in production code. Moreover, as software developers look toward multicore-enabling their legacy codebases, nonlocal variables pose a significant obstacle to software reliability, because they can cause race bugs.

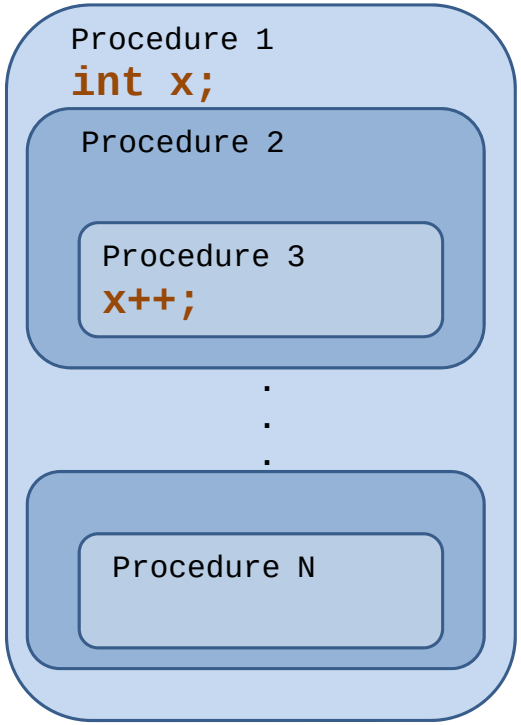
William A. Wulf

Mary Shaw

The lure of nonlocal variables

To begin with, what were Wulf and Shaw concerned about, and why are nonlocal variables nevertheless so prevalent? To be clear, by ***nonlocal*** variable, I mean one nonlocal that is declared outside of the scope in which it is used. A ***global variable*** is a nonlocal variable declared in the outermost program scope.

Wulf and Shaw were concerned that when a variable is used far away from its definition, a local inspection of the code cannot easily determine its meaning. They identified several reasons for eschewing nonlocal variables. Among them, they observed that if a function modifies the nonlocal variable, this side-effect can be difficult to reason about.



In their words, "Untold confusion can result when the consequences of executing a procedure cannot be determined at the site of the procedure call."

The clear alternative to referencing nonlocal variables is to reference local ones, which one can do by passing the variable as an argument to function calls. The problem with this approach is that it leads to ***parameter proliferation*** — long argument lists to functions for passing numerous, frequently used variables. For every global variable referenced in a function, an extra parameter would have to be declared to the function to pass that variable. Not only would that lead to functions with dozens of extra arguments, there also the cost of passing the arguments to consider.

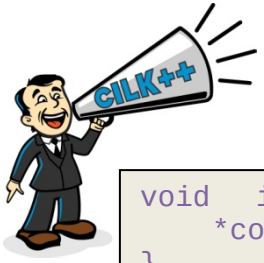
As a practical matter, nonlocal variables are just darn convenient. If I have a code operating on a large data structure, why should I have to pass the data structure to each and every function that operates on the data structure? It's much more convenient to keep the function interfaces clean with fewer parameters and simply refer to the data structure as a global variable or, commonly in object-oriented programming, as a nonlocal member variable. Unfortunately, global and other nonlocal variables can inhibit parallelism by inducing race bugs.

North American Blackout of 2003

Race conditions

Race conditions are the bane of concurrency. Famous race bugs include the Therac-25 radiation therapy machine, which killed three people and injured several others, and the North American Blackout of 2003, which left over 50 million people without power. These pernicious bugs are notoriously hard to find. You can run regression tests in the lab for days without a failure only to discover that your software crashes in the field with regularity. If you're going to multicore-enable your application, you need a reliable way to find and eliminate race conditions.

Different types of race conditions exist depending on the synchronization methodology (*e.g.*, locking, condition variables, etc.) used to coordinate parallelism in the application. Perhaps the most basic of race conditions, and the easiest to understand, is the




```
void   incr   (int   *counter)   {
    *counter++;
}
void main() {
    int x(0);
    cilk_spawn incr (&x);
    incr (&x);
    cilk_sync;
    assert (x == 2);
}
```

www.cilk.com/EVP

"determinacy race," because this kind of race doesn't involve a synchronization methodology at all. A program is ***deterministic*** if it always does the same thing on the same input, no matter how the instructions are scheduled on the multicore computer, and it's ***nondeterministic*** if its behavior might vary from run to run. Often, a parallel program that is intended to be deterministic isn't, because it contains a determinacy race.

In the following examples, we'll assume that the underlying hardware supports the ***sequential consistency*** memory model, where the parallel program execution can be viewed as an interleaving of the steps of the processes, threads, strands, or whatever the abstraction for independent locus of control in the parallel-programming model.

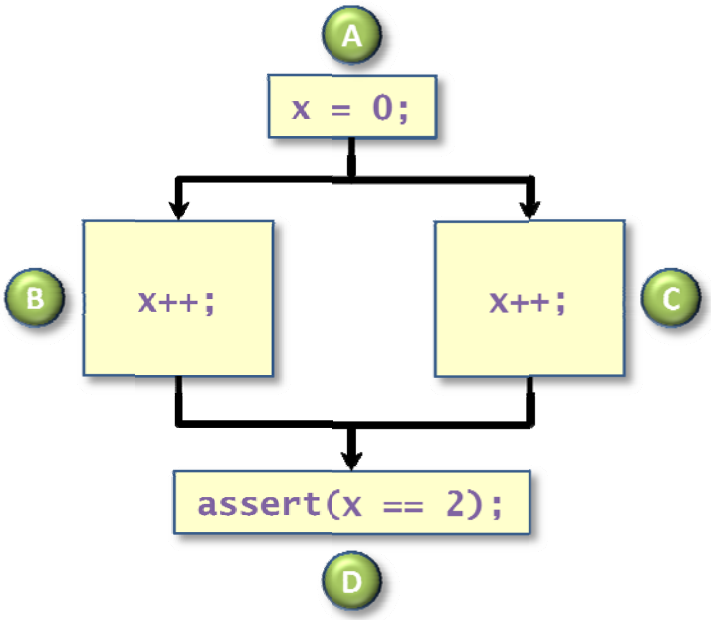
A simple example

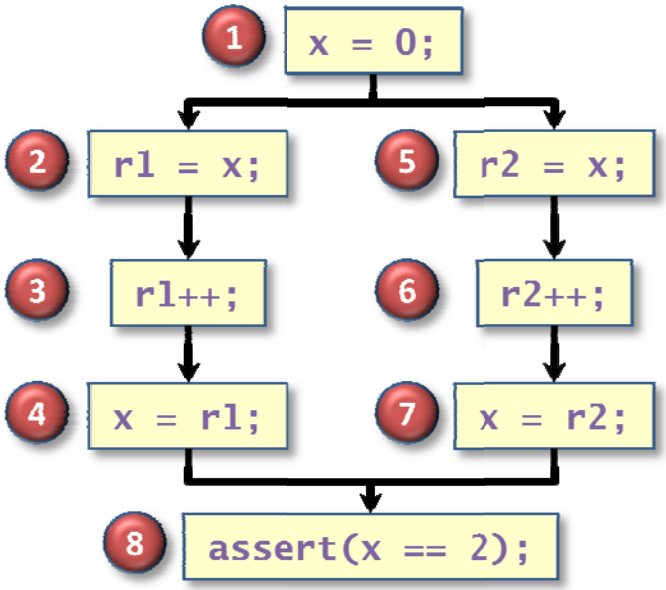
Let's look at an example of a determinacy-race bug, and then we'll define determinacy races more precisely. The Cilk++ code on the left illustrates a determinacy race on a shared variable `x`.

The `cilk_spawn` keyword calls `incr()` but allows control to continue to the following statement. The `cilk_sync` keyword says control shouldn't go past this point until the spawned subroutine has completed. In an ordinary serial execution (equivalent to executing the code with the `cilk_spawn` and `cilk_sync` keywords nulled out), the result is that the value of `x` is increased to `2`. This parallel code has a bug, however, and in a parallel execution, it might sometimes produce `1` for the value of `x`. To understand why, it's helpful to have a clearer understanding of the parallel-program execution.

Using the dag model

We can view the program execution in terms of the dag model of multithreading described in Chapter 2. On the left is a graphical view of the four strands in our simple example using the dag model of multithreading. For convenience, we've collapsed sequences of instructions that don't contain any parallel control, such as `cilk_spawn`





or `cilk_sync`, into ***strands***. Strand A begins with the start of the program and ends at the `cilk_spawn` statement. Two subsequent strands, B and C, are created at the `cilk_spawn` statement: B executes the spawned subroutine `incr()`, and C executes the called subroutine `incr()` on the next line. These two strands join at the `cilk_sync` statement, where Strand D begins. Strand D consists of the instructions from the `cilk_sync` to the end of the program. As the diagram shows, we have $A \prec B$, $A \prec C$, $B \prec D$, $C \prec D$, and by inference, $A \prec D$. We also have $B \parallel C$.

We can now define a determinacy race formally.

> **Definition.** A ***determinacy race*** occurs when two logically parallel instructions access the same location of memory and one of the instructions performs a write.

If S performs the write and S′ performs a read, we call the determinacy race a ***read race***. The behavior of the program depends on whether S′ sees the value before or after S's write. If S′ performs a write, the determinacy race is a ***write race***. In this case, one of the two writes is "lost," and the behavior of the program may depend on which thread wrote last.

Atomicity

The key reason that the example code exhibits a determinacy race is that the `counter++` statement in the definition of `incr()` is not ***atomic***, meaning that it is made up of smaller operations. In the figure on the left, we've broken the increment of the variable `x` into a load from memory into a register, an increment of the register, and then a store of the result back into memory.

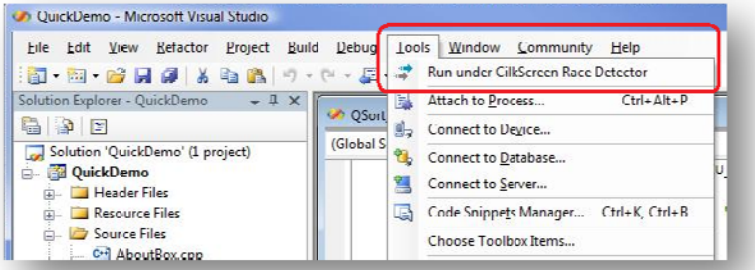
(We've also eliminated a bunch of instructions that don't involve computation on `x`.) Strand A executes the instruction with label 1; Strand B executes the code with labels 2, 3, and 4; Strand C executes the code with labels 5, 6, and 7; and Strand D executes the instruction with label 8.

When the multicore computer executes two strands in parallel, it is as if the various operations making up the strand were interleaved. Now, we can see why the code might operate in a faulty manner. If the instructions are executed in the order (1, 2, 3, 5, 6, 7, 4, 8), Strands B and C both read 0 for the value of x and subsequently both store 1. Thus, x is incremented only once, not twice as we would like. Ouch!

Of course, there are many executions that don't elicit the bug. For example, if the execution order were (1, 2, 3, 4, 5, 6, 7, 8) or (1, 4, 5, 6, 2, 3, 4, 8), we'd be okay. That's the problem with determinacy races: generally, *most* orderings are okay (in this case, any for which B executes before C or vice versa); just some generate improper results (B and C interleave). Consequently, races are notoriously hard to test for. You can run tests for days in the lab and never see a bug, only to release your software and have a customer repeatedly experience the bug (often a catastrophic crash) because of some timing situation in their environment.

Validating determinacy

How can you make sure that your parallel program has no determinacy races? You can try to do it by manual inspection, but that's tedious and error-prone. You have to make sure that every possible ordering of instructions in a parallel execution produces the same behavior. There is an easier way, however. Cilk Arts provides a race-detection tool, called Cilkscreen, which can tell you if your Cilk++ program is free of determinacy races. Moreover, if there is a race, it will locate the offending accesses in the source code, including stack traces. You can check correctness of your code by testing it on serial executions and then use Cilkscreen to verify that there are no determinacy races.



Of course, there are other sources of nondeterminism besides determinacy races. A program that queries time of day, for instance, may behave differently from run to run. Generally, however, we can view these sources of nondeterminism as part of the program input. If you're testing such a program in an ordinary serial-programming environment, you usually dummy up the call to always return a fixed time so that you

can check whether the output produced by the program corresponds to the value expected by your regression suite. You can do the same for a parallel code.

Some programs are correct even with determinacy races. For example, even the example code above might be considered correct if all a programmer cares about is whether x is zero or nonzero. Programs that use locks also typically contain determinacy races that can cause nondeterministic behavior, but the behavior may nevertheless be acceptable. When it isn't acceptable, it's usually because of a special kind of determinacy race, called a ***data race***, which is a topic for another discussion.

Coping with race bugs

There are several ways to deal with race bugs. One way to avoid the data race is to use ***mutual-exclusion locks***, or ***mutexes***, to ensure that only one thread accesses a non-local variable at any point in time. Although locking can "solve" race bugs, lock contention can destroy all parallelism.

Another strategy to eliminate a race bug is to restructure the code to eliminate nonlocal references. Making local copies of the nonlocal variables can remove contention, but at the cost of restructuring program logic. Although the restructured solution incurs some overhead due to parameter passing, it generally produces code with predictably good performance. Unfortunately, however, code restructuring can mean revising the logic of substantial sections of code, and the prospect of refactoring a large codebase is daunting. Should you think you're safe because you don't use recursion or lots of nested function calls, think again! The same problem can arise when the body of a parallel loop accesses variables declared outside the loop.

There's a third way: Cilk++ provides ***hyperobjects*** to mitigate data races on nonlocal variables without the need for locks or code restructuring. Hyperobjects mitigate data races on nonlocal variables without the performance problems endemic to locking or the need for code restructuring — the best of both worlds. The basic idea is that different parallel branches of the computation may see different views of the hyperobject, but combined they provide a single, consistent view.

4. The Folly of Do-It-Yourself Multithreading

OpenMP™

Cilk++

Microsoft .net™

Windows and Linux (and other Unixes) provide API's for creating and manipulating operating system threads using WinAPI threads and POSIX threads (Pthreads), respectively. These threading approaches may be convenient when there's a natural way to functionally decompose an application — for example, into a user-interface thread, a compute thread, a render thread, and so on. (The approach can even create a feeling of "parallelism" on single-processor systems through time-division multiplexing, but this task switching should not be confused with real parallelism on real processing cores.) To many technical "experts" within the company, do-it-yourself (DIY) multithreading directly using the WinAPI or Pthread primitives sounds technically challenging and fun — or at least when the project kicks off. Besides, in the words of Longfellow, won't doing it yourself ensure that it is well done?

The alternative is to program atop a ***concurrency platform*** — an abstraction layer of software that coordinates, schedules, and manages the multicore resources. Concurrency platforms include any of various thread-pool libraries, such as .NET's ThreadPool class; message-passing libraries, such as MPI; data-parallel programming languages, such as NESL, RapidMind, or variants of Fortran since Fortran 90; task-parallel libraries, such as Intel's Threading Building Blocks (TBB) or Microsoft's Task Parallel Library (TPL); or parallel linguistic extensions, such as OpenMP, Cilk, or Cilk++. As can be seen from this sample, some concurrency platforms provide a new language, some extend an existing one, while others are simply implemented as library functions.

Three desirable properties

Although a concurrency platform may provide benefits over DIY multithreading with respect to application performance and software reliability, a strong case against DIY can be made purely on the basis of development time. Indeed, as a rule, one can generally develop robust multithreaded software faster using a concurrency platform than doing it yourself with native threads. The reason is that the DIY strategy makes it hard to obtain three desirable properties of multicore software:

- **Scalability**
- **Code simplicity**
- **Modularity**

A tiny example

```
1   #include <stdio.h>
2   #include <stdlib.h>

3   int fib(int n)
4   {
5     if (n < 2) return n;
6     else {
7       int x = fib(n-1);
8       int y = fib(n-2);
9       return x + y;
10    }
11  }

12  int main(int argc, char *argv[])
13  {
14    int n = atoi(argv[1]);
15    int result = fib(n);
16    printf("Fibonacci of %d is %d.\n", n, result);
17    return 0;
18  }
```

To illustrate the folly of DIY multithreading and its adverse impact on development time, let's look at the simple example of parallelizing a recursive Fibonacci calculation. The `i`th Fibonacci number is the `i`th number (indexed from 0) in the sequence (0, 1, 1, 2, 3, 5, 8, 13, 21, …), where each number is the sum of the previous two. Although this example is tiny and artificial, it illustrates the key issues. On the left is the original code in C/C++.

Incidentally, this algorithm is a terrible way to calculate Fibonacci numbers, since it continually recalculates already computed values and runs in exponential time. Didactically, however, it's short and good enough for our purposes. Just remember that our goal is to parallelize this particular algorithm, not to replace it with a more efficient algorithm for computing Fibonacci numbers.

A version using native threads

On the next page is a Pthreaded version designed to run on 2 processor cores. A WinAPI-threaded implementation would follow similar logic, only differing in the names of library functions. This code is not the most optimized threaded code, but it is fairly typical of what one must do to parallelize the application for 2 processor cores.

The program computes `fib(n)` by handing off the calculation of `fib(n-1)` to a subsidiary thread. As it turns out, on a typical x86 dual-core laptop, the cost of starting up a subsidiary thread is sufficiently great that it's actually slower to use two threads if `n` isn't big enough. A little experimentation led to hardcoding in a threshold of 30 in line 33 for when it's worth creating the subsidiary thread, which leads to about a 1.5 times speedup for computing `fib(40)`.

Although it may seem a little tedious, let's look at the implementation in some detail to examine the logic of this code, because it reveals what DIY programming with native

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3  #include <pthread.h>

 4  int fib(int n)
 5  {
 6    if (n < 2) return n;
 7    else {
 8      int x = fib(n-1);
 9      int y = fib(n-2);
10      return x + y;
11    }
12  }

13  typedef struct {
14    int input;
15    int output;
16  } thread_args;

17  void *thread_func ( void *ptr )
18  {
19    int i = ((thread_args *) ptr)->input;
20    ((thread_args *) ptr)->output = fib(i);
21    return NULL;
22  }
23
24  int main(int argc, char *argv[])
25  {
26    pthread_t thread;
27    thread_args args;
28    int status;
29    int result;
30    int thread_result;
31    if (argc < 2) return 1;
32    int n = atoi(argv[1]);
33    if (n < 30) result = fib(n);
34    else {
35      args.input = n-1;
36      status = pthread_create(&thread,
                              NULL,
                              thread_func,
                              (void*) &args );
        // main can continue executing while the thread executes.
37      result = fib(n-2);
        // Wait for the thread to terminate.
38      pthread_join(thread, NULL);
39      result += args.output;
40    }
41    printf("Fibonacci of %d is %d.\n", n, result);
42    return 0;
43  }
```

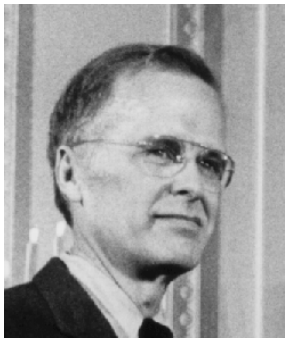
threads is all about. Assuming it is worthwhile to use two threads, we can create the subsidiary thread using the Pthread library function `pthread_create()`. This function takes as arguments a pointer to the function the thread will run after it's created and a pointer to the function's single argument — in this case to `thread_func()` and the struct `args`, respectively. Thus, line 35 marshals the argument `n-1` by storing it into the `input` field of `args`, and line 36 creates the subsidiary thread which calls the wrapper function `thread_func()` on argument `args`. While the subsidiary thread is executing, the main thread goes on to compute `fib(n-2)` in parallel line 37. The subsidiary thread unpacks its argument in line 19, and line 20 stores the result of computing `fib(n-1)` into the `output` field of `args`. When the subsidiary thread completes, as tested for by `pthread_join()` in line 38, the main thread adds the two results together in line 39.

The impact on development time

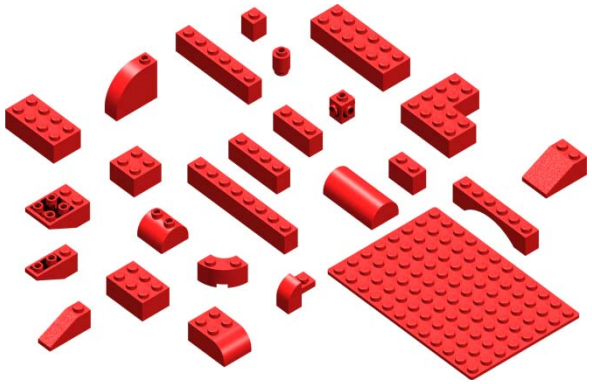
We're now in a position to evaluate how development time might be impacted by coding in a DIY multithreading style.

Scalability

The first thing to note is the lack of scalability. The native-threaded code offers speedup only on a dual-core processor, and it doesn't even attain a factor of 2, because the load isn't balanced between the two threads. Additional coding will be needed to scale to quad-core, and as each generation of silicon doubles the number of cores, even more coding will be required. You might think that you could write the code to use 1000 threads, but unfortunately, the overhead of creating 1000 threads would dominate the running time. Moreover, the system would require memory resources proportional to 1000, even when running on a few cores or just a single core. What's needed is a load-balancing scheduler that manages memory intelligently, and — as luck would have it — that turns out to be one of the services that most concurrency platforms provide.



John W. Backus



Code simplicity

Without a concurrency platform, the effort of thread management can complicate even a simple problem like Fibonacci, harkening back to the days when programmers had to write their own linkage protocols to communicate arguments to called subroutines. That was John Backus's great innovation embodied in the first FORTRAN compiler in 1957. You could call a subroutine without elaborate marshaling of arguments and return values, just by naming them in a functional syntactic form. By contrast, in the native-threaded Fibonacci code, the argument `n-1` must be marshaled into the `args` struct in order to pass it to the subsidiary thread, and then it must be unpacked by the wrapper function `thread_func()` on arrival, much as in pre-FORTRAN caller-callee linkage. DIY multithreading takes a veritable 50-year leap backward! Encapsulating parallel linkage so that it can be expressed in a functional syntactic form is another service that many concurrency platforms provide.

Modularity

Unlike in the serial version, the Fibonacci logic in the native-threaded version is no longer nicely encapsulated in the `fib()` routine. In particular, the identity `fib(n) = fib(n-1) + fib(n-2)` has now infiltrated the main thread. For large applications, a mandated change to the basic serial logic often requires the parallel logic to be modified as well to ensure consistency. You might think that updating both the serial and parallel logic might as much as double developer time, but it's actually much worse in practice. As can be seen from this little example, parallel code can be far more complicated than serial code, and maintaining a DIY multithreaded codebase can cost a factor of 10 or more in developer time over a functionally equivalent serial code. A concurrency platform may provide linguistic support so that the program logic doesn't weasel its way into complex thread-management code.

The bottom line

One can criticize DIY multithreading from the points of view of application performance and software reliability, as well as development time, but the case in terms of development time alone is strong. For example, to obtain good performance and

www.cilk.com/EVP

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <cilk.h>

4   int fib(int n)
5   {
6     if (n < 2) return n;
7     else {
8       int x = cilk_spawn fib(n-1);
9       int y = fib(n-2);
10      cilk_sync;
11      return x + y;
12    }
13  }

14  int cilk_main(int argc, char *argv[])
15  {
16    int n = atoi(argv[1]);
17    int result = fib(n);
18    printf("Fibonacci of %d is %d.\n", n, result);
19    return 0;
20  }
```

ensure reliable software may require extra attention to coding that could also adversely impact development time.

You may now be convinced of the folly of DIY multithreading, but if you're a top-flight developer, you may still harbor thoughts of building your own concurrency platform rather than acquiring one. Before going too far down that path, however, you would be wise to consider the large development efforts that have gone into existing concurrency platforms. Building a concurrency platform from scratch is a mountain to climb. Although many of today's concurrency platforms fall short of offering complete solutions to multicore programming, most do represent a better place to start than does DIY multithreading, and they're improving rapidly.

In summary, parallel programming has earned a reputation for being difficult, and a good deal of that credit owes itself to applications written with native threads. So, if you don't mind that your multicore software is a mess, if you don't mind giving corner offices to the few engineers who have at least some clue as to what's going on, if you don't mind the occasional segment fault when your software is in the field, then DIY multithreading may fit the bill perfectly. ☺ But, if that prospect sounds at least mildly unpleasant, check out one of the concurrency platforms mentioned at the beginning of this chapter and discussed in greater detail in Chapter 5.

To get you started, on the left is a parallelization of the `fib()` code using the Cilk++ concurrency platform. Compare this code to the original, and you judge the impact on development time.

5. Concurrency Platforms

There are a variety of concurrency platforms available today, each with their strengths and weaknesses. (For a broader list, the Wikipedia entry for "parallel programming model" has a nice summary.) Below, we summarize some of the more popular ones.

Thread pools

One of the problems with programming Pthreads or WinAPI threads directly is the overhead associated with creating and destroying them. A thread pool is a strategy for minimizing that overhead and is possibly the simplest concurrency platform. The basic idea of a thread pool is to create a set of threads once and for all at the beginning of the program. When a task is created, it executes on a thread in the pool, returning the thread to the pool when the task is done.

As a practical matter, however, few thread pools are organized in this fashion. The problem is, what happens when a task arrives to find that there are no threads left in the pool? Consequently, most thread pools are implemented so that the newly created tasks are placed into a ***work queue***, and when a thread finishes a task, it fetches a new task to execute from the work queue. If the work queue is empty, it suspends and is woken up when a new task is placed on the work queue. Synchronization using, for example, a mutual-exclusion lock, is necessary to ensure that tasks are removed atomically so that threads cooperate safely when removing tasks from the work queue.

Thread pools are commonly used for Internet servers, where the tasks represent independent client transactions executing on the server. In this context, the main issue is scalability, since the work queue represents a bottleneck in the system. Suppose that the average time to execute a task is x and that the time to remove a task from the queue is y. Then, the number of threads that can productively use the thread pool is at most x/y, assuming each thread has a processing core to run on. More than that, and some threads will be starved waiting on the work queue. For small-scale systems, this limitation is unlikely to matter much, but most thread-pool implementations are not really "future-proof" for this reason.

For application programming, as opposed to server implementation, thread pools pose some concurrency risks. The reason is that the tasks making up an application tend to be dependent on each other. In particular, ***deadlock*** is a significant concern. A deadlock occurs when a set of threads creates a cycle of waiting. For example, suppose that thread 1 holds mutex lock A and is waiting to acquire mutex B, thread 2 is holding

mutex B and is waiting to acquire mutex C, and thread 3 holds lock C and is waiting to acquire mutex A. In this situation, none of the three threads can proceed. Although deadlock is a concern in any asynchronous concurrency platform, thread pools escalate the concern. In particular, a deadlock can occur if all threads are executing tasks that are waiting for another task on the work queue in order to produce a result, but the other task cannot run because all threads are occupied with tasks that are waiting.

Message passing

The high-performance computing (HPC) community has been programming parallel computers for over 20 years, and their *de facto* standard concurrency platform is MPI, a message-passing library. Why not use this tried-and-true technology to multicore enable a commercial codebase? After all, MPI is used to program the world's largest supercomputing clusters. These supercomputing clusters differ from the chip-multiprocessor architecture of x86 multicore chips in a fundamental way: the memory in the system is distributed, not shared. That is, each processor has its own private memory, and if you want to communicate something from one processor to another, you must marshal the data in a buffer and invoke one of the MPI library functions to move the contents. You can't just pass a pointer.

Scientific codes written with MPI have been ported to multicore systems and run quite well. Intuitively, if a code runs well on distributed memory, shared memory won't hurt it. But, MPI is not the solution for multicore-enabling most commercial codebases, because it suffers from many of the same problems as Pthreads and WinAPI threads. Indeed, MPI represents to distributed-memory multiprocessors what Pthreads and WinAPI threads represent to shared-memory multicore systems: an assembly language for programming concurrent programs. (Thus, some might argue whether message-passing actually merits the designation "concurrency platform.") Moreover, absent shared memory, programming using message passing is even more difficult than using native threads.

The abstraction provided by MPI is that processors send and receive messages. The `MPI_Send( )` function is the primitive to send data from one place to another:

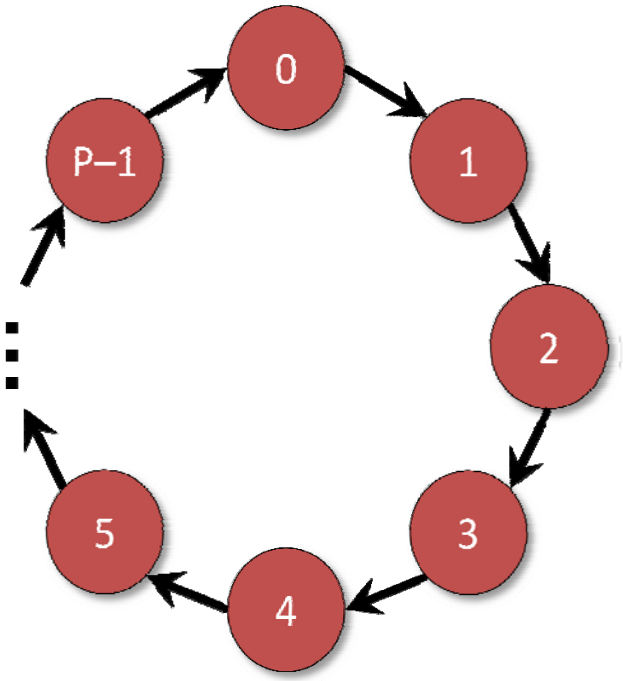
```
int MPI_Send(void *buffer, int count, int datatype, int
dest, int tag, int comm)
```

`buffer` — address of the data to send
`count` — number of data items in the buffer
`datatype` — type of data in the buffer
`dest` — id of the destination processor
`tag` — message type
`comm` — communicator group

The function returns an error code. The corresponding `MPI_Recv()` function receives the data:

```
int MPI_Recv(void *buffer, int count, int datatype, int
source, int tag, int comm, MPI_Status *status)
```

`buffer` — the buffer where the received data should be placed
`count` — number of data items in the buffer
`datatype` — type of data in the buffer
`source` — id of the sending processor
`tag` — message type
`comm` — communicator group
`status` — message status

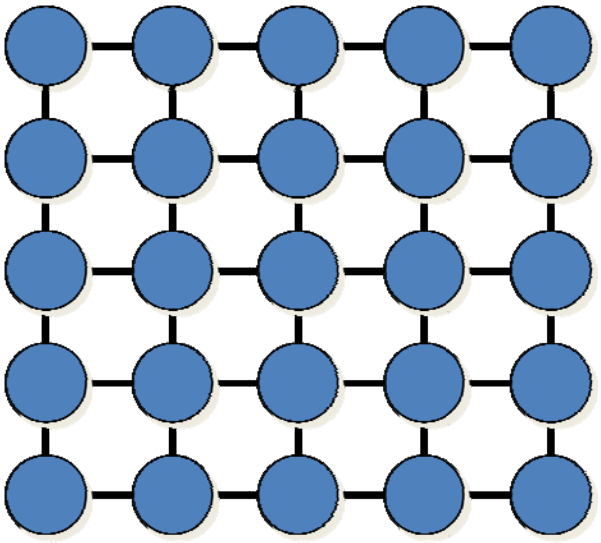


The `MPI_Send()`function ***blocks*** (does not return) until after the message has been received. The `MPI_Recv()` function blocks until a message arrives. Thus, it is important for every `MPI_Send()` to be matched with a corresponding `MPI_Recv()`on the destination processor. Otherwise, the system can deadlock. MPI also provides nonblocking operations, where the programmer assumes the responsibility of explicitly checking to see whether an operation is complete.

It is tedious to use only these simple send/receive primitives to program a large application. To understand why, consider a program in which each processor i simply wishes to send a message to processor i+1 modulo P, the number of processors. That is, the communication pattern is a ring, as shown on the left. One cannot simply write a program in which each processor i executes an `MPI_Send()` to send its message to

processor i+1 (mod P) followed by an `MPI_Recv()` to receive a message from processor i–1 (mod P). The reason is that this program can deadlock, since the `MPI_Send()`'s are not properly matched up with the `MPI_Recv()`'s. Instead, the program must direct the even-numbered processors to execute an `MPI_Send()` followed by an `MPI_Recv()`, while the odd-numbered processors execute an `MPI_Recv()` followed by an `MPI_Send()`, assuming that P is even. If P is odd, additional code is required to handle the boundary case of processor P–1 communicating with processor 0, since they're both even.

The 12-argument library function `MPI_Sendrecv()` allows the programmer to both send a message and receive a message as a single operation, thereby side-stepping the convoluted program logic needed to avoid deadlock. Many other common patterns of communication are special-cased in the library for similar reasons, which explains in some measure why MPI has grown to over 300 functions.



Many scientific-computing applications can be programmed using regular grids in two or more dimensions, as shown on the left. These communication patterns are well supported by MPI, and hence its popularity in the HPC community. If your application exhibits a regular structure, a message-passing concurrency platform may be worth considering, especially if you also want your application to run on distributed-memory clusters. If you have irregular data structures, however, you may find that the development time required to deploy correct message-passing code is prohibitive. Moreover, message passing tends to be heavyweight, and the messages must contain at least kilobytes of data to be efficient, and thus fine-grained applications can suffer greatly from communication overheads.

Data-parallel languages

For scientists working on vectors and matrices, data-parallel languages offer a natural way to express element-wise parallelism on aggregate data. Data-parallel languages were originally developed for the HPC industry, where they continue to be in widespread use, predominantly on shared-memory supercomputers. For example, the Fortran family of languages has, since Fortran 90, allowed parallelizable loops to be

replaced by single ***array statements***. For example, if A, B, and C are N x N matrices, Fortran 90 allows the simple statement

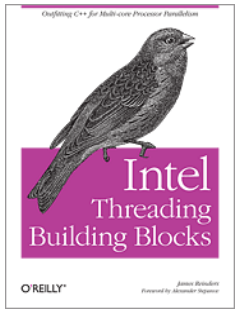
```
C = A + B
```

to replace the following Fortran 77 loop:

```
DO I = 1, N
    DO J = 1, N
        C(I,J) = A(I,J) + B(I,J)
    END DO
END DO
```

Rather than having to decode the Fortran 77-style loop to determine whether it is parallelizable, the compiler can directly parallelize the array statement on the available processor cores. Recent Fortrans also provide a variety of array-sectioning expressions that allow subarrays to be manipulated as a unit.

A big advantage of the data-parallel approach is that the control logic is serial: conceptually there is only one program counter. Thus, a race condition happens within a single array statement, making it easier to localize within the code, and many array operations, such as the elementwise addition above, can be easily inspected to be race free. Potential disadvantages of data-parallel languages include that legacy code must be rewritten, that lack of data locality makes it hard to exploit modern cache hierarchies, and that it is difficult to express irregular parallelism. This latter disadvantage is mitigated somewhat in "nested" data-parallel languages, such as the NESL language developed by Carnegie Mellon University's Scandal Project. Nested data-parallel languages allow arrays to be nested and different functions to be applied elementwise in parallel. In addition, for application areas where data is regular, such as low-level video and image processing, the data-parallel approach seems to have a niche. RapidMind, which provides a C++-based data-parallel environment, also seems to have had some success targeting graphical processing units (GPU's), which are notoriously difficult to program.



Arch Robison

Intel's Threading Building Blocks

Intel's Threading Building Blocks (TBB) is an open-source C++ template library developed by Intel for writing task-based multithreaded applications. The library includes data structures and algorithms that allow a programmer to avoid tedious low-level implementation details. TBB provides an abstraction layer for the programmer, allowing logical sequences of operations to be treated as tasks that are allocated to individual cores dynamically by the library's runtime engine.

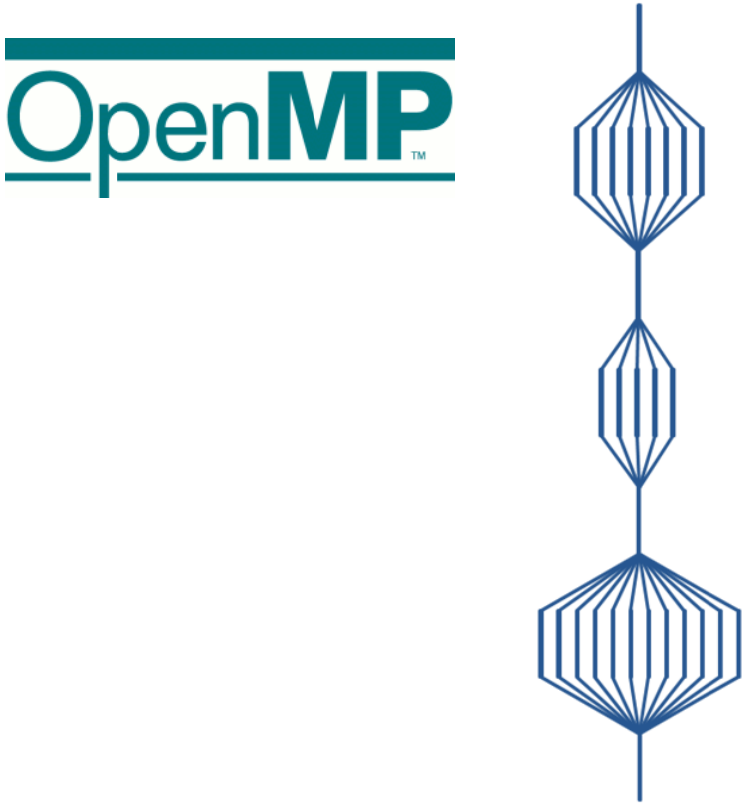
TBB is strictly a library and provides no linguistic support by design. As Arch Robison, lead developer of TBB argues, "Though new languages and extensions are attractive, they raise a high barrier to adoption in the near term …." Consequently, they argue that TBB can be integrated into legacy C++ environments comparatively easily. The programmer breaks an application into multiple tasks, which are scheduled using a "work-stealing" scheduler such as was pioneered by the MIT Cilk project. TBB provides templates for common parallel programming patterns, such as loops, software pipelining, and nested parallelism.

The following example shows how one might square every element of an array using TBB's `parallel_for` template function:

```
#include "tbb/blocked_range.h"
class SqChunk {
    float *const local_a;
public:
    void operator()(const blocked_range<size_t>& x) const {
        float *a = local_a;
        for(size_t i=x.begin(); i!=x.end(); ++i)
            a[i] *= a[i];
    }
    SqChunk(float a[]) :
        local_a(a)
    {}
};
void Square(float a[], size_t n) {
    parallel_for(blocked_range<size_t>(0,n,1000), SqChunk(a));
}
```

This code operates by breaking the input array into chunks and then running each chunk in parallel. The `SqChunk` class provides an object that processes a chunk by serially squaring every element in a given range. The `blocked_range` template class is provided by TBB, which specifies an iteration space of `0` to `n-1` broken into chunks of size `1000`. The `parallel_for` takes this chunking specification and runs the `SqChunk` on each chunk in parallel.

OpenMP



OpenMP (Open Multi-Processing) is an open-source concurrency platform that supports multithreaded programming through Fortran and C/C++ language ***pragmas*** (compiler directives). OpenMP compilers are provided by several companies, including Intel, Microsoft, Sun Microsystems, IBM, Hewlett-Packard, Portland Group, and Absoft, and it is also supported the Gnu gcc compiler. By inserting pragmas into the code, the programmer identifies the sections of code that are intended to run in parallel.

One of OpenMP's strengths is parallelizing loops such as are found in many numerical applications. For example, consider the following C++ OpenMP code snippet which sums the corresponding elements of two arrays:

```
#pragma omp parallel for
for (i=0; i<n; ++i) {
    c[i] = a[i] + b[i];
}
```

The pragma indicates to the compiler that the iterations of the loop that follows can run in parallel. The loop specification must obey a certain set of patterns in order to be parallelized, and OpenMP does not attempt to determine whether there are dependencies between loop iterations. If there are, the code has a race. An advantage in principle to the pragma strategy is that the code can run as ordinary serial code if the pragmas are ignored. Unfortunately, some of the OpenMP directives for managing

memory consistency and local copies of variables affect the semantics of the serial code, compromising this desirable property unless the code avoids these pragmas.

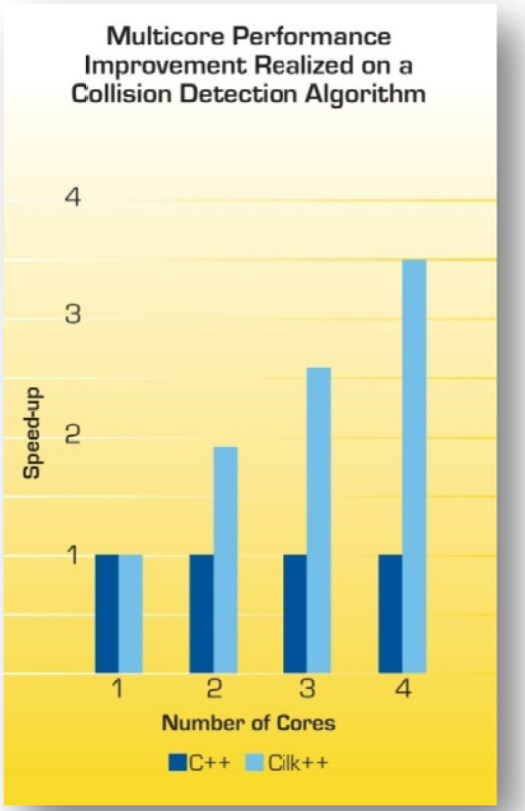
OpenMP schedules the loop iterations using a strategy called ***work sharing***. In this model, parallel work is broken into a collection of chunks which are automatically farmed out to the various processors when the work is generated. OpenMP allows the programmer to specify a variety of strategies for how the work should be distributed. Since work-sharing induces communication and synchronization overhead whenever a parallel loop is encountered, the loop must contain many iterations in order to be worth parallelizing.

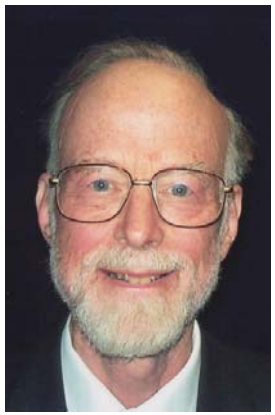
Although OpenMP was designed primarily to support a single level of loop parallelization, alternating between serial sections and parallel sections, as illustrated on the left, some implementations of OpenMP support nested parallelism. The work-sharing scheduling strategy is often not up to the task, however, because it is sometimes difficult to determine at the start of a nested loop how to allocate thread resources. In particular, when nested parallelism is turned on, it is common for OpenMP applications to "blow out" memory at runtime because of the inordinate space demands. The latest generation OpenMP compilers have started to address this problem, however.

Cilk++

Cilk++, from Cilk Arts, extends C++ into the multicore realm without sacrificing serial semantics. Cilk++ was inspired by the award-winning MIT Cilk project, which pioneered key ideas in multithreading, including the "work-stealing" strategy for scheduling. The Cilk++ concurrency platform provides the following:

1. **Three Cilk++ keywords**: The programmer inserts Cilk++ keywords into a serial C++ application to expose parallelism. The resulting Cilk++ source retains the serial semantics of the original C++ code. Consequently, programmers can still develop their software in the familiar serial domain using their existing tools. Serial debugging and regression testing remain unchanged.





C.A.R. Hoare

2. **The Cilk++ compiler**: The Cilk++ compiler recognizes the three Cilk++ keywords and generates a multicore-enabled application. Because the Cilk++ compiler is an extension of an industry standard compiler, it supports the native features and optimizations provided by the base compiler.

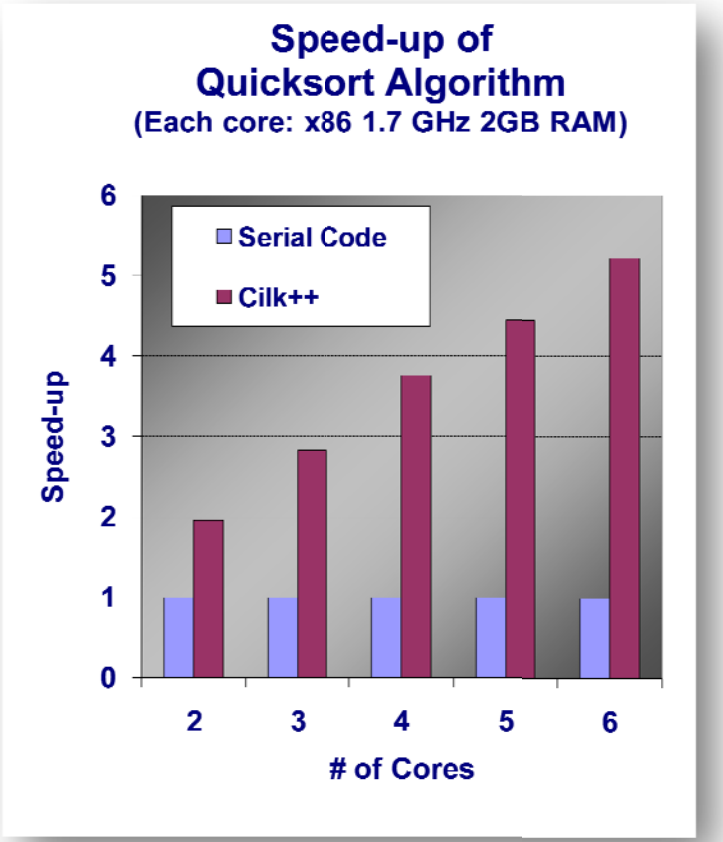
3. **Cilk++ hyperobjects**: Global and other nonlocal variables can inhibit parallelism by inducing race conditions. Cilk hyperobjects are an innovative construct that side-steps races that might otherwise be created by parallel accesses to global variables.

4. **The Cilk++ runtime system**: The Cilk++ RTS links with the binary executable of the Cilkified program. The RTS contains a proprietary "work-stealing" scheduler that is provably efficient.

5. **Cilkscreen**: The Cilkscreen™ race detector is used to test a Cilk++ binary program to ensure its parallel correctness. Cilkscreen is mathematically guaranteed to find all race conditions in a Cilk++ program execution.

As an example, let's see how to use Cilk++ to multicore-enable the following C++ program which implements Tony Hoare's classical quicksort algorithm:

```
template <typename Iter>
void qsort(Iter begin, Iter end) {
// Sort the range between begin and end-1.
       typedef typename std::iterator_traits<Iter>::value_type T;
   if (begin != end) {
       --end;      // Exclude pivot (last element) from partition
       Iter middle = std::partition(begin, end,
                        std::bind2nd(std::less<T>(),*end));
       using std::swap;
       swap(*end, *middle);    // Move pivot to middle
       qsort(begin, middle);
       qsort(++middle, ++end); // Exclude pivot and restore end
   }
}
```



To parallelize this code using Cilk++, we insert two Cilk++ keywords into the code:

```
template <typename Iter>
void qsort(Iter begin, Iter end) {
// Sort the range between begin and end-1.
       typedef typename std::iterator_traits<Iter>::value_type T;
   if (begin != end) {
       --end;     // Exclude pivot (last element) from partition
       Iter middle = std::partition(begin, end,
                      std::bind2nd(std::less<T>(),*end));
       using std::swap;
       swap(*end, *middle);    // Move pivot to middle
       cilk_spawn qsort(begin, middle);
       qsort(++middle, ++end); // Exclude pivot and restore end
       cilk_sync;
   }
}
```

The Cilk++ keyword `cilk_spawn` indicates that the named ***child*** function — in this case, the recursive quicksort on the lower part of the partition — can execute in parallel with the ***parent*** caller. After calling the recursive quicksort on the upper part of the partition, we execute a `cilk_sync`, which prevents control from passing that point until all the spawned children have completed. Because the `qsort()` routine is recursive, this code generates a tree of parallel activity. (Cilk++ also provides a `cilk_for` keyword which can be used to parallelize loops.)

The graph on the left shows the speedup obtained. A `cilk_spawn`/`cilk_sync` is over 450 times faster than a Pthread `create/exit` — less than 4 times slower than an ordinary C++ function call. As a result, Cilk++ overhead typically measures just a few percent on a single processor, and for the `qsort()` program, it is less than 1 percent.

Cilk++ obtains this kind of speedup because the RTS contains a powerful ***work-stealing*** scheduler. Unlike a work-sharing scheduler, which immediately disburses parallel work to processors when the work is generated and thereby incurs communication and synchronization overheads, with a work-stealing scheduler, each processor posts

www.cilk.com/EVP

generated work to its own local queue. When a processor runs out of work, it steals work from another processor's queue. This strategy incurs no overhead for communication and synchronization when there is ample parallelism. Only when a processor runs out of work does the execution incur these overheads, which can be mathematically proved to be small if there is sufficient parallelism in the application. Moreover, unlike most work-sharing schedulers, the Cilk++ work-stealing scheduler guarantees that the stack space used by the parallel code when run on P processors is no more than P times the stack space of a one-processor execution.

The C++ code that is obtained by deleting the `cilk_spawn` and `cilk_sync` keywords from the Cilk++ code and replacing `cilk_for` by `for` is called the ***serialization*** of the Cilk++ code. The C++ serialization of a Cilk++ program is always a legal interpretation of the Cilk++ code. Moreover, a Cilk++ program running on a single processing core executes identically to the C++ serialization. Most other concurrency platforms do not provide this simplifying property.

One advantage of this approach is that conventional debugging tools can be used to ensure the serial correctness of Cilk++ programs. To test for parallel correctness, the programmer runs the Cilkscreen race detector, which is guaranteed to find any discrepancies between the execution of the serialization and a parallel execution. In particular, if there is any way that the scheduling of a Cilk++ code could differ from its C++ serialization, Cilkscreen guarantees to locate the determinacy race responsible.

6. Twenty Questions to Ask When Going Multicore

To help you survive the multicore revolution, and determine the best concurrency platform for your unique needs, we have compiled a set of key questions to ask — some regarding your application, others regarding your potential solution. No one-size-fits-all solution exists. Your optimal choice must be driven by your unique project's requirements. We've organized the questions in terms of the three legs of the

multicore software triad: application performance, software reliability, and development time.

Application performance

1. Does the concurrency platform allow me to measure the parallelism I've exposed in my application?
2. Does the concurrency platform address response-time bottlenecks, or just offer more throughput?
3. Does application performance scale up linearly as cores are added, or does it quickly reach diminishing returns?
4. Is my multicore-enabled code just as fast as my original serial code when run on a single processor?
5. Does the concurrency platform's scheduler load-balance irregular applications efficiently to achieve full utilization?
6. Will my application "play nicely" with other jobs on the system, or do multiple jobs cause thrashing of resources?
7. What tools are available for detecting multicore performance bottlenecks?

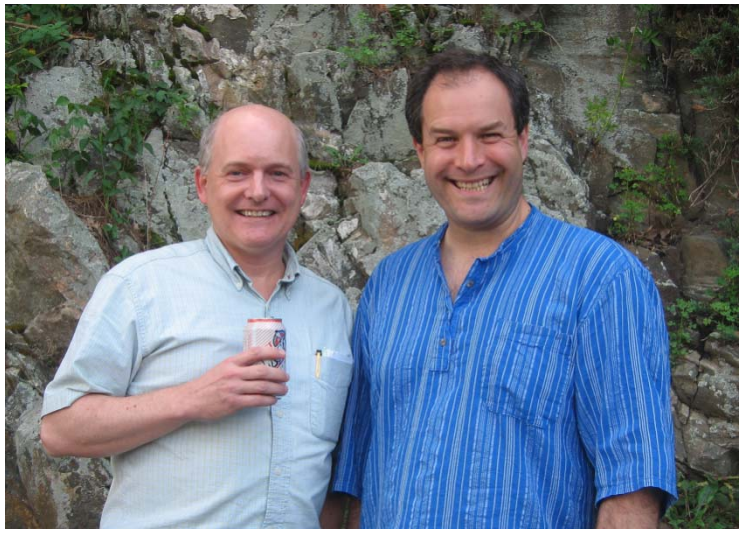
Software reliability

8. How much harder is it to debug my multicore-enabled application than to debug my original application?
9. Can I use my standard, familiar debugging tools?
10. Are there effective debugging tools to identify and localize parallel-programming errors, such as data-race bugs?
11. Must I use a parallel debugger even if I make an ordinary serial programming error?
12. What changes must I make to my release-engineering processes to ensure that my delivered software is reliable?
13. Can I use my existing unit tests and regression tests?

Development time

14. To multicore-enable my application, how much logical restructuring of my application must I do?
15. Can I easily train programmers to use the concurrency platform?
16. Can I maintain just one code base, or must I maintain a serial and parallel versions?
17. Can I avoid rewriting my application every time a new processor generation increases the core count?
18. Can I easily multicore-enable ill-structured and irregular code, or is the concurrency platform limited to data-parallel applications?
19. Does the concurrency platform properly support modern programming paradigms, such as objects, templates, and exceptions?
20. What does it take to handle global variables in my application?

About the Authors



Charles E. Leiserson and Ilya Mirman

Charles E. Leiserson is a founder of Cilk Arts, Inc., and Professor of Computer Science and Engineering at MIT. His research work on parallel computing spans 30 years. Charles enjoys family, skiing, and Diet Moxie.

Ilya Mirman is VP of Marketing at Cilk Arts, Inc., and has spent the past decade building, marketing, and supporting software tools for engineers. Prior to that, he designed high-speed lasers for long-haul telecom networks. Ilya enjoys jamming with friends, freelance political photography, and San Pellegrino mineral water.

CILK ARTS

About Cilk Arts

Cilk Arts is a company OF software engineers, building a product FOR software engineers. We celebrate beauty in engineering, and are jazzed about the opportunity to help build a community that changes the future of programming. Inspired by 15 years of award-winning research at MIT, Cilk Arts is building Cilk++, a cross-platform solution that offers the easiest, quickest, and most reliable way to maximize application performance on multicore processors.

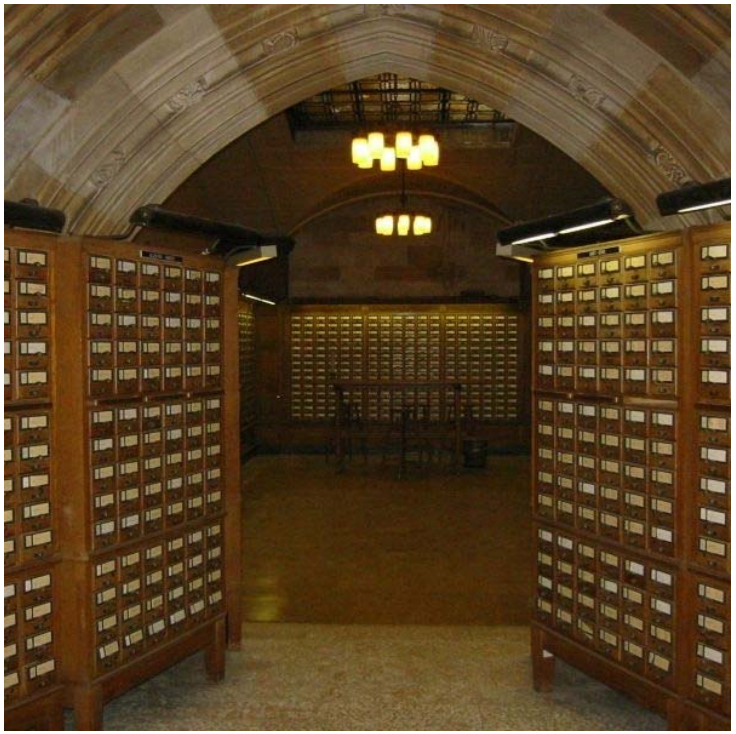
Cilk++ provides a simple set of extensions for C++, coupled with a powerful runtime system for multicore-enabled applications. Cilk++ enables rapid development, testing, and deployment of multicore applications. Cilk++ solves the two large problems facing the software industry as a result of the multicore revolution:

1. Enabling today's mainstream programmers to develop multithreaded (or parallel) applications; and
2. Providing a smooth path to multicore for legacy applications that otherwise cannot easily leverage the performance capabilities of multicore processors.

On our website, www.cilk.com, we have pulled together resources of interest to folks interested in multicore programming — including links to commercial solutions and open-source projects, technical notes, and links to blogs and other multicore-related sites.

- Multicore Programming Resources
- Subscribe to Multicore Blog
- Join the Early Visibility Program
- Cilk++ Solution Overview and Product Demonstration

Index



Amdahl's Law 3
array statement 23
atomic 11
atomicity 11
blocking function 21
child 28
chip multiprocessor 1
`cilk_for` 28
`cilk_spawn` 10, 28
`cilk_sync` 10, 28
Cilk++ 26
Cilkscreen 12
concurrency platform 14
core 1
critical path 6
dag model of multithreading 4
data race 13
data-parallel language 22
deadlock 19
determinacy race 11
deterministic program 9
global variable 8
high-performance computing (HPC) 20
hyperobject 13
linear speedup 5
lock 13
message passing 20
Moore's Law 1
multicore software triad 2
mutual-exclusion lock (mutex) 13
nondeterministic program 10
nonlocal variable 8
OpenMP 25
parallel 4
parallelism 6
parameter proliferation 9
parent 28
perfect linear speedup 5
POSIX threads (pthreads) 14
pragmas 25
processing core 1
quicksort 27
race condition 9
read race 11
sequential consistency 10
serialization 29
span 6
Span Law 6
speedup 5
strand 10
superlinear speedup 5
TBB 24
thread 14
thread pool 19
Threading Building Blocks 24
WinAPI threads 14
work 5
Work Law 5
work queue 19
work sharing 26
work stealing 28
write race 11

www.cilk.com/EVP

Multicore-enable your C++ Application:

Join the Cilk++ Early Visibility Program!

www.cilk.com/EVP

Multicore Challenges		CILK Arts Solution		
Development Time	• Will you be forced to redesign your application? • How will you get your product out in time? • Where will you find enough parallel programming talent?	**SHRINK Development Time**		✔ Minimal application changes ✔ Can be learned in hours by a C++ programmer ✔ Seamless path forward (and backward)
Application Performance	• Can you achieve best-in-class performance? • Will your solution scale with the number of cores?	**BOOST Application Performance**		✔ Minimal overhead on a single-core ✔ Linear scaling as cores are added ✔ Adaptive load balancing
Software Reliability	• Can you debug and test your multi-core application? • Can you assure that there are no race conditions?	**ENSURE Software Reliability**		✔ Multithreaded version guaranteed as reliable as the original ✔ No fundamental change to release engineering ✔ Automatically detect race conditions