Optional Prereading

We've included:

- Chapter 2 from TBB book
- Chapter 1 from SPP book
- Glossary from SPP book

Chapter 2 from James' 2007 book on TBB has been often recommended as a good introduction to parallelism. We recommend you read it prior to August 4.

You may also want to read more about "why shift to multicore?" and more on programming models. We've included Chapter 1 from the 2012 SPP book for you to enjoy.

Finally, terminology can be challenging – so we've included the Glossary from the SPP book for your enjoyment.

Jackson Marusarz. Technical Consulting Engineer. Jackson is a consulting engineer with Intel's Developer Products Division. He is currently working on developer tools in the Technical Computing, Analyzers, and Runtimes group. His areas of expertise include parallel programming and performance analysis and tuning for both serial and multi-threaded applications. He is the main contributor to the Intel® VTune™ Amplifier XE tuning guides and is currently working on developing tuning methodologies with a focus on microarchitectural bottleneck analysis. Jackson regularly presents online Webinars and works closely with developers locally and remotely through trainings and performance analysis engagements.

James Reinders. Parallel Programming Evangelist. James is involved in multiple engineering, research and educational efforts to increase use of parallel programming throughout the industry. He joined Intel Corporation in 1989, and has contributed to numerous projects including the world's first TeraFLOP/s supercomputer (ASCI Red) and the world's first TeraFLOP/s microprocessor (Intel® Xeon Phi™ coprocessor). James been an author on numerous technical books, including VTune™ Performance Analyzer Essentials (Intel Press, 2005), Intel® Threading Building Blocks (O'Reilly Media, 2007), Structured Parallel Programming (Morgan Kaufmann, 2012), Intel® Xeon Phi™ Coprocessor High Performance Programming (Morgan Kaufmann, 2013), and Multithreading for Visual Effects (A K Peters/CRC Press, 2014). James is working on a project to publish a book of programming examples featuring Intel Xeon Phi programming scheduled to be published in late 2014.

Outfitting C++ for Multi-core Processor Parallelism

Intel Threading Building Blocks

O'REILLY®

James Reinders

Foreword by Alexander Stepanov

CHAPTER 2

Thinking Parallel

This chapter is about how to "Think Parallel." It is a brief introduction to the mental discipline that helps you make a program parallelizable in a safe and scalable manner. Even though Intel Threading Building Blocks does much of the work that traditional APIs require the programmer to do, this kind of thinking is a fundamental requirement to writing a good parallel program.

This is the place in the book that defines terms such as *scalability* and provides the motivation for focusing on these concepts. The topics covered in this chapter are decomposition, scaling, threads, correctness, abstraction, and patterns.

If you don't already approach every computer problem with parallelism in your thoughts, this chapter should be the start of a new way of thinking. If you are already deeply into parallel programming, this chapter can still show how Threading Building Blocks deals with the concepts.

How should we think about parallel programming? This is now a common question because, after decades in a world where most computers had only one central processing unit (CPU), we are now in a world where only "old" computers have one CPU. Multi-core processors are now the norm. Parallel computers are the norm. Therefore, every software developer needs to Think Parallel.

Today, when developers take on a programming job, they generally think about the best approach before programming. We already think about selecting the best algorithm, implementation language, and so on. Now, we need to think about the inherent parallelism in the job first. This will, in turn, drive algorithm and implementation choices. Trying to consider the parallelism after everything else is not Thinking Parallel, and will not work out well.

Elements of Thinking Parallel

Threading Building Blocks was designed to make expressing parallelism much easier by abstracting away details and providing strong support for the best ways to program for parallelism. Here is a quick overview of how Threading Building Blocks addresses the topics we will define and review in this chapter:

Decomposition
: Learning to decompose your problem into concurrent tasks (tasks that can run at the same time).

Scaling
: Expressing a problem so that there are enough concurrent tasks to keep all the processor cores busy while minimizing the overhead of managing the parallel program.

Threads
: A guide to the technology underlying the concurrency in programs—and how they are abstracted by Threading Building Blocks so that you can just focus on your tasks.

Correctness
: How the implicit synchronization inherent in Threading Building Blocks helps minimize the use of *locks*. If you still must use locks, there are special features for using the Intel Thread Checker to find *deadlocks* and *race conditions*, which result from errors involving locks.

Abstraction and patterns
: How to choose and utilize algorithms, from Chapters 3 and 4.

Caches
: A key consideration in improving performance. The Threading Build Blocks task scheduler is already tuned for caches.

Intuition
: Thinking in terms of tasks that can run at the same time (concurrent tasks), data decomposed to minimize conflicts among tasks, and recursion.

In everyday life, we find ourselves thinking about parallelism. Here are a few examples:

Long lines
: When you have to wait in a long line, you have undoubtedly wished there were multiple shorter (faster) lines, or multiple people at the front of the line helping serve customers more quickly. Grocery store checkout lines, lines to get train tickets, lines to buy coffee, and lines to buy books in a bookstore are examples.

Lots of repetitive work

When you have a big task to do, which many people could help with at the same time, you have undoubtedly wished for more people to help you. Moving all your possessions from an old dwelling to a new one, stuffing letters in envelopes for a mass mailing, and installing the same software on each new computer in your lab are examples.

The point here is simple: parallelism is not unknown to us. In fact, it is quite natural to think about opportunities to divide work and do it in parallel. It just might seem unusual for the moment to program that way. Once you dig in and start using parallelism, you will Think Parallel. You will think first about the parallelism in your project, and only then think about coding it.

Decomposition

When you think about your project, how do you find the parallelism?

At the highest level, parallelism exists either in the form of data on which to operate in parallel, or in the form of tasks to execute concurrently. And these forms are *not* mutually exclusive.

Data Parallelism

Data parallelism (Figure 2-1) is easy to picture. Take lots of data and apply the same transformation to each piece of the data. In Figure 2-1, each letter in the data set is capitalized and becomes the corresponding uppercase letter. This simple example shows that given a data set and an operation that can be applied element by element, we can apply the same task concurrently to each element. Programmers writing code for supercomputers love this sort of problem and consider it so easy to do in parallel that it has been called *embarrassingly parallel*. A word of advice: if you have lots of data parallelism, do not be embarrassed—take advantage of it and be very happy. Consider it *happy parallelism*.

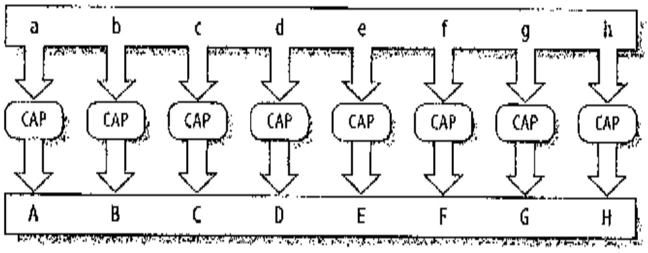

Figure 2-1. Data parallelism

Task Parallelism

Data parallelism is eventually limited by the amount of data you want to process, and your thoughts will then turn to task parallelism (Figure 2-2). Task parallelism means lots of different, independent tasks that are linked by sharing the data they consume. This, too, can be *embarrassingly parallel*. Figure 2-2 uses as examples some mathematical operations that can each be applied to the same data set to compute values that are independent. In this case, the average value, the minimum value, the binary OR function, and the geometric mean of the data set are computed.

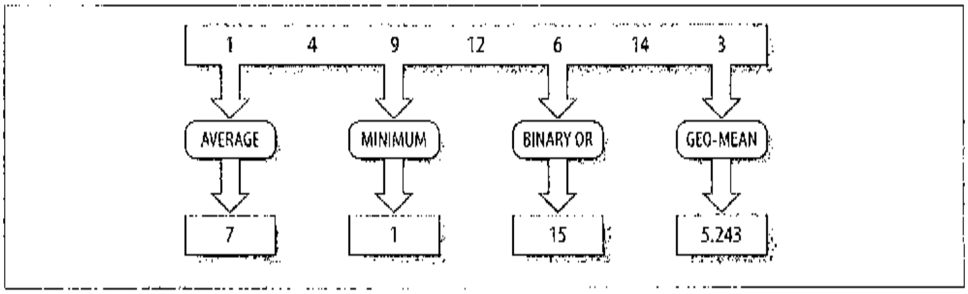

Figure 2-2. Task parallelism

Pipelining (Task and Data Parallelism Together)

Pure task parallelism is harder to find than pure data parallelism. Often, when you find task parallelism, it's a special kind referred to as *pipelining*. In this kind of algorithm, many independent tasks need to be applied to a stream of data. Each item is processed by stages as they pass through, as shown by the letter A in Figure 2-3. A stream of data can be processed more quickly if you use a pipeline because different items can pass through different stages at the same time, as shown in Figure 2-4. A pipeline can also be more sophisticated than other processes: it can reroute data or skip steps for chosen items. Automobile assembly lines are good examples of pipelines; materials flow through a pipeline and get a little work done at each step (Figure 2-4).

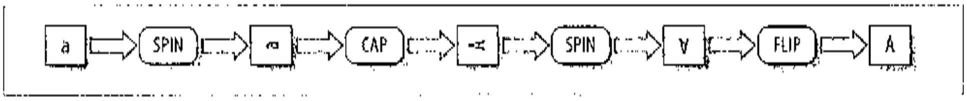

Figure 2-3. Pipeline

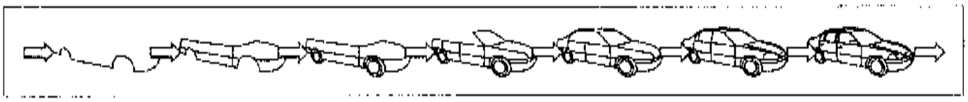

Figure 2-4. A pipeline in action with data flowing through it

Mixed Solutions

Consider the task of folding, stuffing, sealing, addressing, stamping, and mailing letters. If you assemble a group of six people for the task of stuffing many envelopes, you can arrange each person to specialize in and perform one task in a pipeline fashion (Figure 2-5). This contrasts with data parallelism, where you divide the supplies and give a batch of everything to each person (Figure 2-6). Each person then does all the steps on his collection of materials as his task.

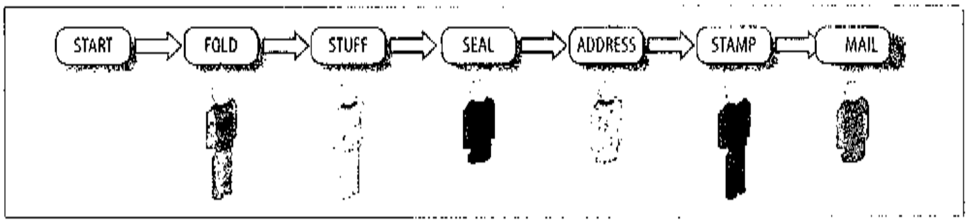

Figure 2-5. Pipelining—each person has a different job

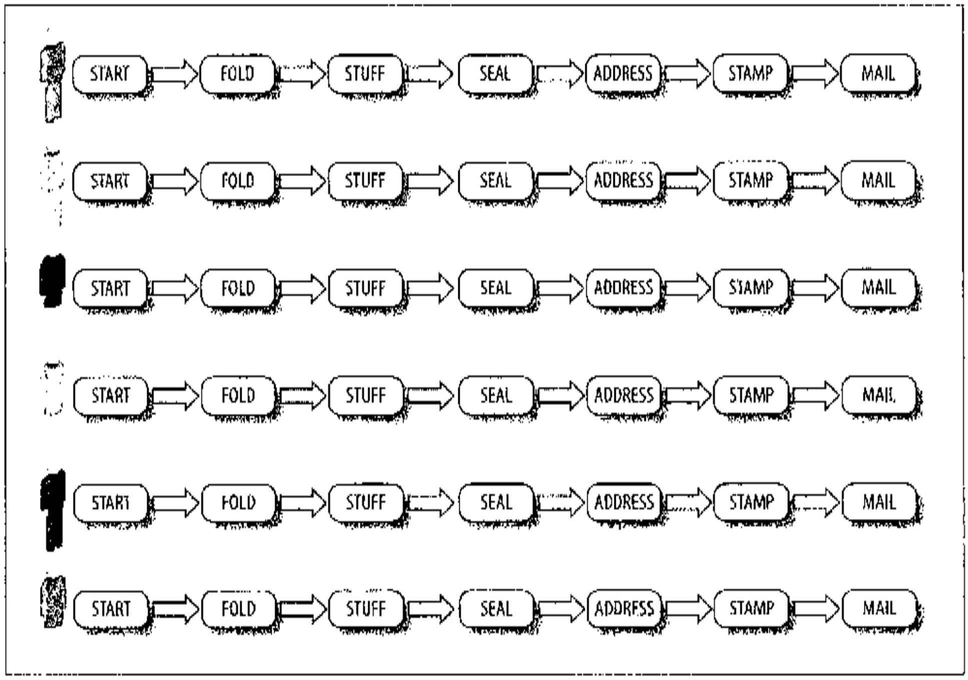

Figure 2-6. Data parallelism—each person has the same job

Figure 2-6 is clearly the right choice if every person has to work in a different location far from each other. That is called *coarse-grained* parallelism because the interactions between the tasks are infrequent (they only come together to collect envelopes, then leave and do their task, including mailing). The other choice shown in Figure 2-5 is known as *fine-grained* parallelism because of the frequent interactions (every envelope is passed along to every worker in various steps of the operation).

Neither extreme tends to fit reality, although sometimes they may be close enough to be useful. In our example, it may turn out that addressing an envelope takes enough time to keep three people busy, whereas the first two steps and the last two steps require only one person on each pair of steps to keep up. Figure 2-7 illustrates the steps with the corresponding size of the work to be done. The resulting pipeline (Figure 2-8) is really a hybrid of data and task parallelism.

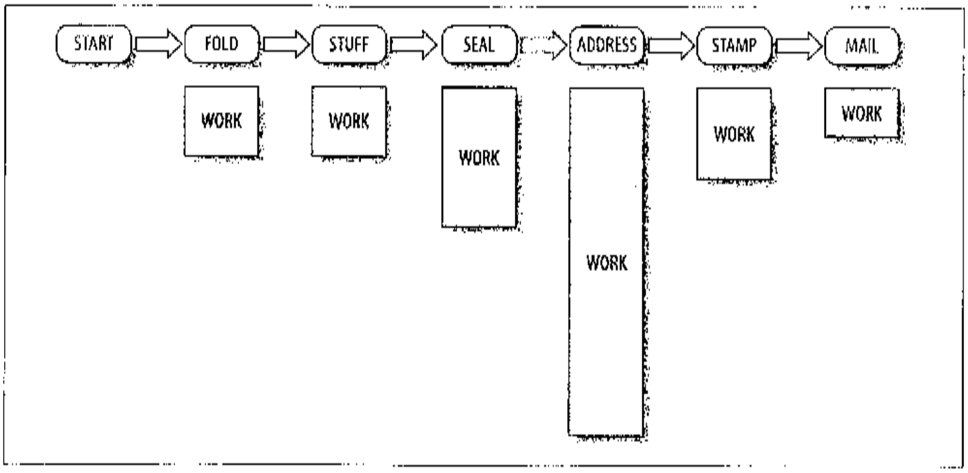

Figure 2-7. Unequal tasks are best combined or split to match people

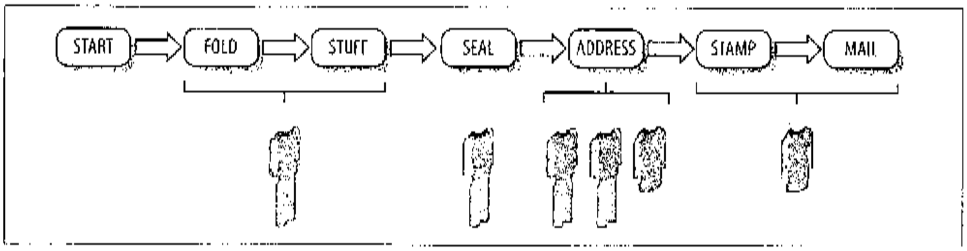

Figure 2-8. Because tasks are not equal, assign more people to the addressing task

Achieving Parallelism

Coordinating people around the job of preparing and mailing the envelopes is easily expressed by the following two conceptual steps:

1. Assign people to tasks (and feel free to move them around to balance the workload).
2. Start with one person on each of the six tasks, but be willing to split up a given task so that two or more people can work on it together.

The six tasks are folding, stuffing, sealing, addressing, stamping, and mailing. We also have six people (resources) to help with the work. That is exactly how Threading Building Blocks works best: you define tasks and data at a level you can explain and then split or combine data to match up with resources available to do the work.

The first step in writing a parallel program is to consider where the parallelism is. Many textbooks wrestle with task and data parallelism as though there were a clear choice. Threading Building Blocks allows any combination of the two that you express.

If you are lucky, your program will be cleanly data-parallel only. To simplify this work, Threading Building Blocks requires you only to specify tasks and how to split them. For a completely data-parallel task, in Threading Building Blocks you will define one task to which you give all the data. That task will then be split up automatically to use the available hardware parallelism. The implicit synchronization will often eliminate the need for using locks to achieve synchronization.

People have been exploring decomposition for decades, and some patterns have emerged. We'll cover this more later when we discuss design patterns for parallel programming.

Scaling and Speedup

The scalability of a program is a measure of how much *speedup* the program gets as you add more and more processor cores. Speedup is the ratio of the time it takes to run a program without parallelism versus the time it runs in parallel. A speedup of 2X indicates that the parallel program runs in half the time of the sequential program. An example would be a sequential program that takes 34 seconds to run on a one-processor machine and 17 seconds to run on a quad-core machine.

As a goal, we would expect that our program running on two processor cores should run faster than the program running on one processor core. Likewise, running on four processor cores should be faster than running on two cores.

We say that a program does not *scale* beyond a certain point when adding more processor cores no longer results in additional speedup. When this point is reached, it is common for performance to fall if we force additional processor cores to be used. This is because the overhead of distributing and synchronizing begins to dominate. Threading Building Blocks has some algorithm templates which use the notion of a *grain size* to help limit the splitting of data to a reasonable level to avoid this problem. Grain size will be introduced and explained in detail in Chapters 3 and 4.

As Thinking Parallel becomes intuitive, structuring problems to scale will become second nature.

How Much Parallelism Is There in an Application?

The topic of how much parallelism there is in an application has gotten considerable debate, and the answer is "it depends."

It certainly depends on the size of the problem to be solved and on the ability to find a suitable algorithm to take advantage of the parallelism. Much of this debate previously has been centered on making sure we write efficient and worthy programs for expensive and rare parallel computers. The definition of size, the efficiency required, and the expense of the computer have all changed with the emergence of multi-core processors. We need to step back and be sure we review the ground we are standing on. The world has changed.

Amdahl's Law

Gene Amdahl, renowned computer architect, made observations regarding the maximum improvement to a computer system that can be expected when only a portion of the system is improved. His observations in 1967 have come to be known as *Amdahl's Law*. It tells us that if we speed up *everything* in a program by 2X, we can expect the resulting program to run 2X faster. However, if we improve the performance of only half the program by 2X, the overall system improves only by 1.33X. Amdahl's Law is easy to visualize. Imagine a program with five equal parts that runs in 500 seconds, as shown in Figure 2-9. If we can speed up two of the parts by 2X and 4X, as shown in Figure 2-10, the 500 seconds are reduced to only 400 and 350 seconds, respectively. More and more we are seeing the limitations of the portions that are not speeding up through parallelism. No matter how many processor cores are available, the serial portions create a barrier at 300 seconds that will not be broken (see Figure 2-11).

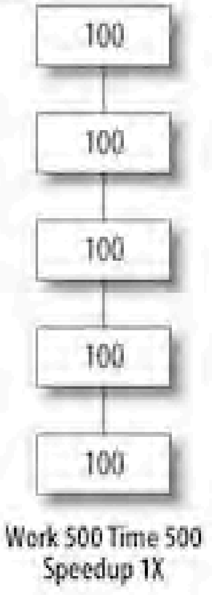

Figure 2-9. Original program without parallelism

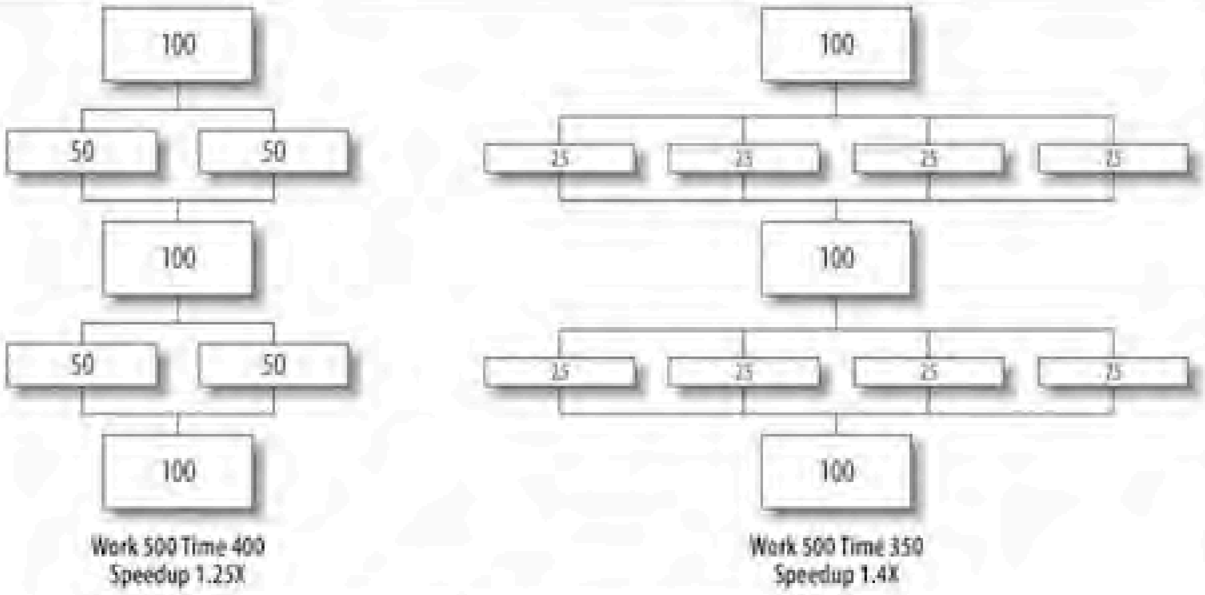

Figure 2-10. Progress on adding parallelism

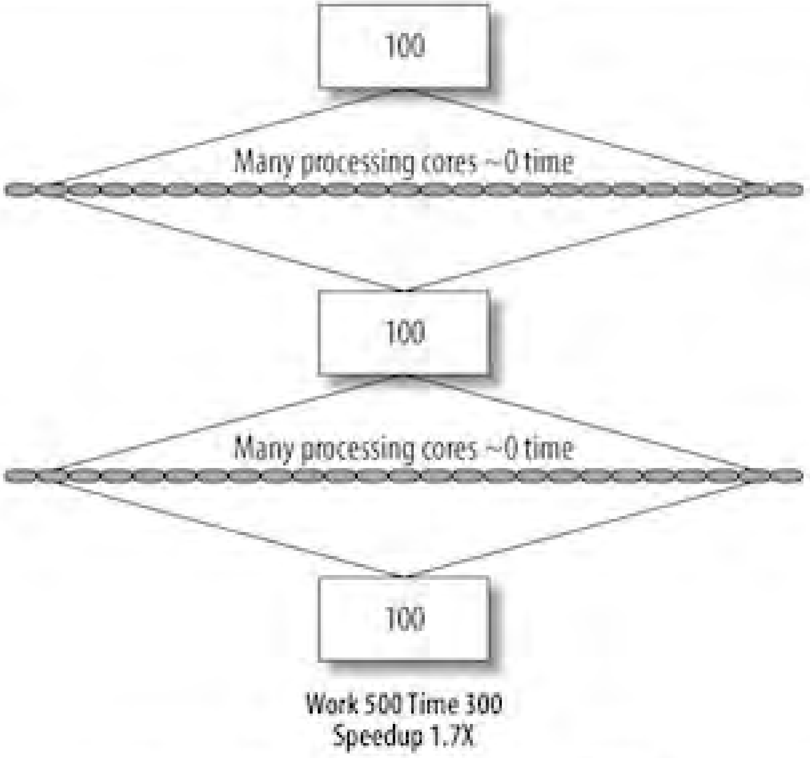

Figure 2-11. Limits according to Amdahl's Law

Parallel programmers have long used Amdahl's Law to predict the maximum speedup that can be expected using multiple processors. This interpretation ultimately tells us that a computer program will never go faster than the sum of the parts that do *not* run in parallel (the serial portions), no matter how many processors we have.

Many have used Amdahl's Law to predict doom and gloom for parallel computers, but there is another way to look at things that shows much more promise.

Gustafson's observations regarding Amdahl's Law

Amdahl's Law views programs as fixed, while we make changes to the computer. But experience seems to indicate that as computers get new capabilities, applications change to take advantage of these features. Most of today's applications would not run on computers from 10 years ago, and many would run poorly on machines that are just five years old. This observation is not limited to obvious applications such as games; it applies also to office applications, web browsers, photography software, DVD production and editing software, and Google Earth.

More than two decades after the appearance of Amdahl's Law, John Gustafson, while at Sandia National Labs, took a different approach and suggested a reevaluation of Amdahl's Law. Gustafson noted that parallelism is more useful when you observe that workloads grow as computers become more powerful and support programs that do more work rather than focusing on a fixed workload. For many problems, as the problem size grows, the work required for the parallel part of the problem grows faster than the part that cannot be parallelized (the so-called serial part). Hence, as the problem size grows, the serial fraction decreases and, according to Amdahl's Law, the scalability improves. So we can start with an application that looks like Figure 2-9, but if the problem scales with the available parallelism, we are likely to see the advancements illustrated in Figure 2-12. If the sequential parts still take the same amount of time to perform, they become less and less important as a percentage of the whole. The algorithm eventually reaches the conclusion shown in Figure 2-13. Performance grows at the same rate as the number of processors, which is called *linear* or *order of n scaling*, denoted as O(*n*).

Even in our example, the efficiency of the program is still greatly limited by the serial parts. The efficiency of using processors in our example is about 40 percent for large numbers of processors. On a supercomputer, this might be a terrible waste. On a system with multi-core processors, one can hope that other work is running on the computer concurrently to use the processing power our application does not use. This new world has many complexities. In any case, it is still good to minimize serial code, whether you take the "glass half empty" view and favor Amdahl's Law or you lean toward the "glass half full" view and favor Gustafson's observations.

Both Amdahl's Law and Gustafson's observations are correct. The difference lies in whether you want to make an existing program run faster with the same workload or you envision working on a larger workload. History clearly favors programs getting more complex and solving larger problems. Gustafson's observations fit the historical evidence better. Nevertheless, Amdahl's Law is still going to haunt us as we work today to make a single application work faster on the same benchmark. You have to look forward to see a brighter picture.

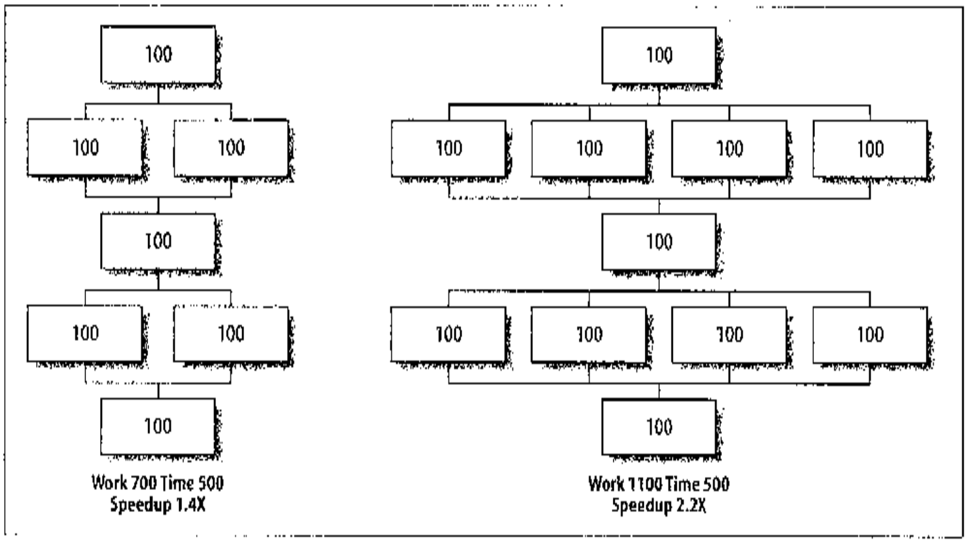

Figure 2-12. Scale the workload with the capabilities

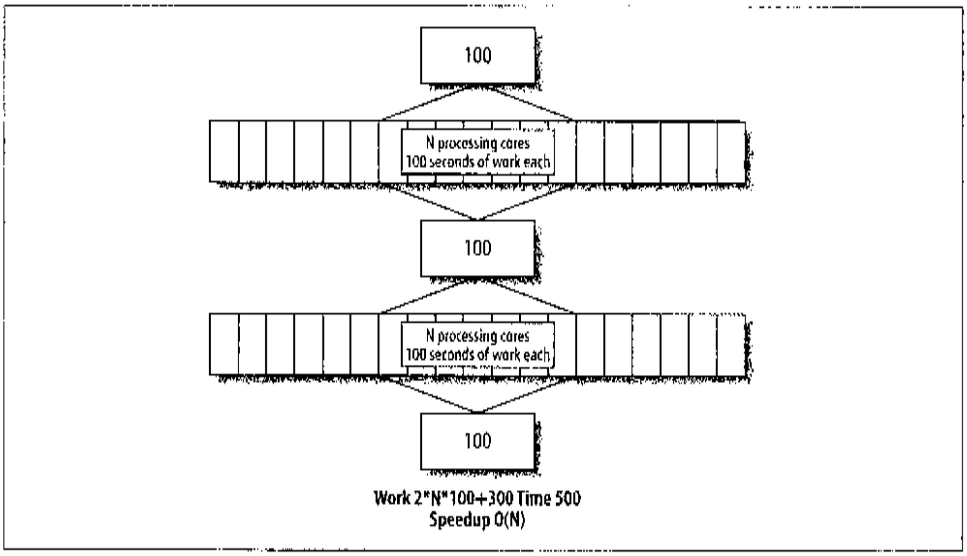

Figure 2-13. Gustafson saw a path to scaling

The value of parallelism is easier to prove if you are looking forward than if you assume the world is not changing.

Making today's application run faster by switching to a parallel algorithm without expanding the problem is harder than making it run faster on a larger problem. The value of parallelism is easier to prove when we are not constrained to speeding up an application that already works well on today's machines.

The scalability of an application comes down to increasing the work done in parallel and minimizing the work done serially. Amdahl motivates us to reduce the serial portion, whereas Gustafson tells us to consider larger problems, where the parallel work is likely to increase relative to the serial work.

Some have defined scaling that requires the problem size to grow as *weak scaling*. It is ironic that the term *embarrassingly parallel* is commonly applied to other types of scaling. Because almost all true scaling happens only when the problem size scales with the parallelism available, we should call that just *scaling*. We can apply the term *embarrassing scaling* to scaling that occurs without growth in the size. As with *embarrassing parallelism*, if you have *embarrassing scaling*, take advantage of it and do not be embarrassed.

What did they really say?

Here is what Amdahl and Gustafson actually said in their famous papers, which have generated much dialog ever since:

> ...the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude. —Amdahl, 1967

> ...speedup should be measured by scaling the problem to the number of processors, not by fixing the problem size. —Gustafson, 1988

Combining these ideas, you can conclude that the value of parallelism is easier to prove if you are looking forward than if you assume the world is not changing. If we double the number of cores every couple of years, we should be working to double the amount of work we want our computer to do every couple of years as well.

Serial versus parallel algorithms

One of the truths in programming is this: the best serial algorithm is seldom the best parallel algorithm, and the best parallel algorithm is seldom the best serial algorithm.

This means that trying to write a program that runs well on a system with one processor core, and also runs well on a system with a dual-core processor or quad-core processor, is harder than just writing a good serial program or a good parallel program.

Supercomputer programmers know from practice that the work required in concurrent tasks grows quickly as a function of the problem size. If the work grows faster than the sequential overhead (e.g., communication, synchronization), you can fix a program that scales poorly just by increasing the problem size. It's not uncommon at all to take a program that won't scale much beyond 100 processors and scale it nicely to 300 or more processors just by doubling the size of the problem.

To be ready for the future, write parallel programs and abandon the past. That's the simplest and best advice to offer. Writing code with one foot in the world of efficient single-threaded performance and the other foot in the world of parallelism is the hardest job of all.

What Is a Thread?

If you know what a *thread* is, skip ahead to the section "Safety in the Presence of Concurrency." It's important to be comfortable with the concept of a thread, even though the goal of Intel Threading Building Blocks is to abstract away thread management. Fundamentally, you will still be constructing a threaded program and you will need to understand the implications of this underlying implementation.

All modern operating systems are *multitasking* operating systems that typically use a *preemptive* scheduler. *Multitasking* means that more than one program can be active at a time. You may take it for granted that you can have an email program and a web browser program running at the same time. Yet, not that long ago, this was not the case.

A *preemptive* scheduler means the operating system puts a limit on how long one program can use a processor core before it is forced to let another program use it. This is how the operating system makes it appear that your email program and your web browser are running at the same time when only one processor core is actually doing the work.

Generally, each process (program) runs relatively independent of other processes. In particular, the memory where your program variables will reside is completely separate from the memory used by other processes. Your email program cannot directly assign a new value to a variable in the web browser program. If your email program can communicate with your web browser—for instance, to have it open a web page from a link you received in email—it does so with some form of communication that takes much more time than a memory access.

Breaking a problem into multiple processes and using *only* a restricted, mutually agreed-upon communication between them has a number of advantages. One of the advantages is that an error in one process will be less likely to interfere with other processes. Before multitasking operating systems, it was much more common for a single program to be able to crash the entire machine. Putting tasks into processes, and limiting interaction with other processes and the operating system, has greatly added to system stability.

Multiple processes work with coarse-grained parallelism where the tasks to be done do not require frequent *synchronization*. You can think of synchronization as the computer equivalent of people working on different parts of the same job and stopping occasionally to talk, to hand over finished work to another person, and perhaps to get work from another person. Parallel tasks need to coordinate their work as well. The more fine-grained their parallelism is, the more time is spent communicating between tasks. If parallelism is too fine, the amount of communication between tasks can become unreasonable.

Therefore, all modern operating systems support the subdivision of processes into multiple *threads* of execution. Threads run independently, like processes, and no thread knows what other threads are running or where they are in the program unless they synchronize explicitly. The key difference between threads and processes is that the threads within a process share all the data of the process. Thus, a simple memory access can accomplish the task of setting a variable in another thread.

Each thread has its own *instruction pointer* (a register pointing to the place in the program where it is running) and *stack* (a region of memory that holds subroutine return addresses and local variables for subroutines), but otherwise a thread shares its memory. Even the stack memory of each thread is accessible to the other threads, though when they are programmed properly, they don't step on each other's stacks.

Threads within a process that run independently but share memory have the obvious benefit of being able to share work quickly, because each thread has access to the same memory as the other threads in the same process. The operating system can view multiple threads as multiple processes that have essentially the same permissions to regions of memory.

Programming Threads

A process usually starts with a single thread of execution and is allowed to request that more threads be started. Threads can be used to logically break down a program into multiple tasks, such as a user interface and a main program. Threads are also useful for programming for parallelism, such as with multi-core processors.

Many questions arise once you start programming to use threads. How should you divide and assign tasks to keep each available processor core busy? Should you create a thread each time you have a new task, or should you create and manage a pool of threads? Should the number of threads depend on the number of cores? What should you do with a thread running out of tasks?

These are important questions for the implementation of multitasking, but that doesn't mean you as the application programmer should answer them. They detract from the objective of expressing the goals of your program. Likewise, assembly language programmers once had to worry about memory alignment, memory layout, stack pointers, and register assignments. Languages such as Fortran and C were created to abstract away those important details and leave them to be solved with compilers and libraries. Similarly, today we seek to abstract away thread management so that programmers can express parallelism directly.

Threading Building Blocks takes care of all thread management so that programmers can express parallelism directly with tasks.

A key notion of Threading Building Blocks is that you should break up the program into many more tasks than there are processors. You should specify as much parallelism as practical and let Threading Building Blocks choose how much of that parallelism is actually exploited.

Safety in the Presence of Concurrency

When code is written in such a way that it cannot be run in parallel without the concurrency causing problems, it is said *not* to be *thread-safe*. Even with the abstraction that Threading Building Blocks offers, the concept of thread safety is essential. You need to understand how to build a program that is thread-safe, which essentially means that each function can be invoked by more than one thread at the same time.

Single-threaded programs contain only one flow of control, so their parts need not be reentrant or thread-safe. In multithreaded programs, the same functions and the same resources may be utilized concurrently by several flows of control. Code written for multithreaded programs must therefore be reentrant and thread-safe.

Any function that maintains a persistent state between invocations requires careful writing to ensure it is thread-safe. In general, functions should be written to have no side effects so that concurrent use is not an issue. In cases where global side effects—which might range from setting a single variable to creating or deleting a file—do need to occur, the programmer must be careful to call for *mutual exclusion*, ensuring that only one thread at a time can execute the code that has the side effect, and that other threads are excluded from that section of code.

Be sure to use thread-safe libraries. All the libraries you use should be reviewed to make sure they are thread-safe. The C++ library has some functions inherited from C that are particular problems because they hold internal state between calls, specifically `asctime`, `ctime`, `gmtime`, `localtime`, `rand`, and `strtok`. Be sure to check the documentation if you need to use these functions to see whether thread-safe versions are available. The C++ Standard Template Library (STL) container classes are in general *not* thread-safe (hence, some of the containers defined by Threading Building Blocks are not thread-safe either).

Mutual Exclusion and Locks

You need to think about wherher concurrent accesses to the same resources will occur in your program. The resource with which you will most often be concerned is data held in memory, but you also need to think about files and I/O of all kinds.

The best policy is to decompose your problem in such a way that synchronization is implicit instead of explicit. You can achieve this by breaking up the tasks so that they can work independently, and the only synchronization that occurs is waiting for all the tasks to be completed at the end.

Instead of locks, which are shown in Figure 2-14, you can use a small set of operations that the system guarantees to be *atomic*. An atomic operation is equivalent to an instruction that cannot be interrupted.

When explicit synchronization and atomic operations are insufficient, locks needs to be used. Chapter 7 covers the various options for mutual exclusion.

Consider a program with two threads. We start with X = 44. Thread A executes X = X + 10. Thread B executes X = X - 12. If we add locking (Figure 2-14) so that only Thread A or Thread B can execute its statement at a time, we end up with X = 42. If both threads try to obtain a lock at the same time, one will be excluded and will have to wait before the lock is granted. Figure 2-14 shows how long Thread B might have to wait if it requested the lock at the same time as Thread A but did not get the lock because Thread A had it first.

Thread A	Thread B	Value of X
LOCK (X)	*(wait)*	44
Read X (44)	*(wait)*	44
add 10	*(wait)*	44
Write X (54)	*(wait)*	54
UNLOCK (X)	*(wait)*	54
	LOCK (X)	54
	Read X (54)	54
	subtract 12	54
	Write X (42)	42
	UNLOCK (X)	42

Figure 2-14. Predictable outcome due using mutual exclusion

Without the locks, a race condition exists and at least two more results are possible: X = 32 or X = 54. X = 42 can still occur as well (Figure 2-15). Three results are now possible because each statement reads X, does a computation, and writes to X. Without locking, there is no guarantee that a thread reads the value of X before or after the other thread writes a value.

Thread A	Thread B	Value of X
Read X (44)		44
add 10	Read X (44)	44
Write X (54)	subtract 12	54
	Write X (32)	32

RACE – A first, B second

Thread A	Thread B	Value of X
	Read X (44)	44
	subtract 12	44
	Write X (32)	32
Read X (32)		32
add 10		32
Write X (42)		42

DESIRED

Thread A	Thread B	Value of X
	Read X (44)	44
Read X (44)	subtract 12	44
add 10	Write X (32)	32
Write X (54)		54

RACE – B first, A second

Figure 2-15. Results of race condition (no mutual exclusion)

Correctness

The biggest challenge of learning to Think Parallel is understanding correctness as it relates to concurrency. *Concurrency* means you have multiple threads of control that are active at one time. The operating system is going to schedule those threads in a number of ways. Each time the program runs, the precise order of operations will potentially be different. Your challenge as a programmer is to make sure that every legitimate way the operations in your concurrent program can be ordered will still lead to the correct result. A high-level abstraction such as Threading Building Blocks helps a great deal, but there are a few issues you have to grapple with on your own: potential variations in results when programs compute results in parallel, and new types of programming bugs when locks are used incorrectly.

Computations done in parallel often get different results than the original sequential program. Round-off errors are the most common surprise for many programmers when a program is modified to run in parallel. You should expect numeric results to vary slightly when computations are changed to run in parallel. For example, computing `(A+B+C+D)` as `((A+B)+(C+D))` enables `A+B` and `C+D` to be computed in parallel, but the final sum may be slightly different from other evaluations such as `(((A+B)+C)+D)`. Even the parallel results can differ from run to run, depending on the order of the operations.

A few types of program failures can happen only in a parallel program because they involve the coordination of tasks. These failures are known as *deadlocks* and *race conditions*. Although Threading Building Blocks simplifies programming so as to reduce the chance for such failures, they are still quite possible. Multithreaded programs can be *nondeterministic* as a result, which means the same program with the same input can follow different execution paths each time it is invoked. When this occurs, failures do not repeat consistently and debugger intrusions can easily change the failure, thereby making debugging frustrating, to say the least.

Tracking down and eliminating the source of unwanted nondeterminism is not easy. Specialized tools such as the Intel Thread Checker help, but the first step is to understand these issues and try to avoid them.

There is also another very common problem when moving from sequential code to parallel code: getting different results because of subtle changes in the order in which work is done. Some algorithms may be unstable, whereas others simply exercise the opportunity to reorder operations that are considered to have multiple correct orderings.

Here are two key errors in parallel programming:

Deadlock

Deadlock occurs when at least two tasks wait for each other and each will not resume until the other task proceeds. This happens easily when code requires the acquisition of multiple locks. If Task A needs Lock R and Lock X, it might get Lock R and then try to get Lock X. Meanwhile, if Task B needs the same two locks but grabs Lock X first, we can easily end up with Task A wanting Lock X while holding Lock R, and Task B waiting for Lock R while it holds only Lock X. The resulting impasse can be resolved only if one task releases the lock it is holding. If neither yields, deadlock occurs and the tasks are stuck forever.

Solution

Use implicit synchronization to avoid the need for locks. In general, avoid using locks, especially multiple locks at one time. Acquiring a lock and then invoking a function or subroutine that happens to use locks is often the source of multiple lock issues. Because access to shared resources must sometimes occur, the two most common solutions are to acquire locks in a certain order (always A and then B, for instance) or to release all locks whenever a lock cannot be acquired and begin again.

Race conditions

A race condition occurs when multiple tasks read from and write to the same memory without proper synchronization. The "race" may finish correctly sometimes and therefore complete without errors, and at other times it may finish incorrectly. Figure 2-15 illustrates a simple example with three different possible outcomes due to a race condition.

Race conditions are less catastrophic than deadlocks, but more pernicious because they don't necessarily produce obvious failures and yet can lead to corrupted data: an incorrect value being read or written. The result of some race conditions can be a state that is not legal because a couple of threads may each succeed in updating half their state (multiple data elements).

Solution

Manage shared data in a disciplined manner using the synchronization mechanisms described in Chapter 7 to ensure a correct program. Avoid low-level methods based on locks because it is so easy to get things wrong.

Explicit locks should be a last effort. In general, the programmer is better off using the synchronization implied by the algorithm templates and task scheduler when possible. For instance, use parallel_reduce instead of creating your own with shared variables. The join operation in parallel_reduce is guaranteed not to run until the subproblems it is joining are completed.

Abstraction

When writing a program, choosing an appropriate level of abstraction is important. Few programmers use assembly language anymore. Programming languages such as C and C++ have abstracted away the low-level details. Hardly anyone misses the old programming method.

Parallelism is no different. You can easily get caught up in writing code that is too low-level. Raw thread programming requires you to manage threads, which is time-consuming and error-prone.

Programming in Threading Building Blocks offers an opportunity to avoid thread management. This will result in code that is easier to create, easier to maintain, and more elegant. However, it does require thinking of algorithms in terms of what work can be divided and how data can be divided.

Patterns

Mark Twain once observed, "The past does not repeat itself, but it does rhyme." And so it is with computer programs: code may not be reused over and over without change, but patterns do emerge.

Condensing years of parallel programming experience into patterns is not an exact science. However, we can explain parallel programming approaches in more detail than just talking about task versus data decomposition.

Object-oriented programming has found value in the Gang of Four (Gamma, Helm, Johnson, and Vlissides) and their landmark work, *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison Wesley). Many credit that book with bringing more order to the world of object-oriented programming. The book gathered the collective wisdom of the community and boiled it down into simple "patterns" with names, so people could talk about them.

A more recent book, *Patterns for Parallel Programming*, by Mattson et al. (Addison Wesley), has similarly collected the wisdom of the parallel programming community. Its point is that the field of parallel programming has moved from chaos to established practice. Experts use common tricks and have their own language to talk about these tricks. With these patterns in mind, programmers can quickly get up to speed in this new field, just as object-oriented programmers have done for years with the famous Gang of Four book.

Patterns for Parallel Programming is a serious and detailed look at how to approach parallelism. Tim Mattson often lectures on this topic, and he helped me summarize how the patterns relate to Threading Building Blocks. This will help connect the concepts of this book with the patterns for parallelism that his book works to describe.

Mattson et al. propose that programmers need to work through four design spaces when moving from first thoughts to a parallel program (Table 2-1 summarizes these explanations):

Finding concurrency
: This step was discussed earlier in this chapter, in the section "Decomposition." Threading Building Blocks simplifies finding concurrency by encouraging you to find one or more tasks without worrying about mapping them to hardware threads. For some algorithms (e.g., `parallel_for`), you will supply an iterator that determines how to make a task split in half when the task is considered large enough. In turn, Threading Building Blocks will then divide large data ranges repeatedly to help spread work evenly among processor cores. Other algorithms, such as `tbb::pipeline`, help express the opportunity to create lots of tasks differently. The key for you is to express parallelism in a way that allows Threading Building Blocks to create many tasks.

Algorithm structures
: This step embodies your high-level strategy. Figure 2-16 shows an organizational view for decompositions. In the "Decomposition" section, Figure 2-4 illustrated a Pipeline and Figure 2-2 illustrated Task Parallelism. Threading Building Blocks is a natural fit for these two patterns. Threading Building Blocks also excels at recursion because of its fundamental design around recursion, so the patterns of Divide and Conquer and Recursive Data are easily handled. The Event-Based Coordination pattern is a poor fit for parallelism because it is unstructured and unpredictable. In some cases, using the task scheduler of Threading Building Blocks directly may be an option.

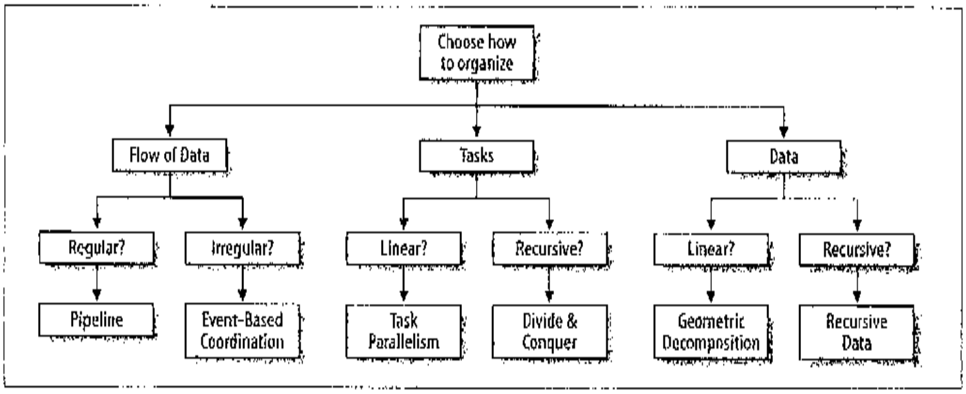

Figure 2-16. A view on organization

Supporting structures

This step involves the details for turning our algorithm strategy into actual code. For the traditional parallel programmer, these issues are critical and have an impact that reaches across the entire parallel programming process. But Threading Building Blocks offers enough support for this design pattern that the programmer rarely has to be concerned with it.

Implementation mechanisms

This step includes thread management and synchronization. Threading Building Blocks handles all the thread management, leaving you free to worry only about tasks at a higher level of design. The design encourages implicit synchronization so as to reduce the need for explicit locking. Chapter 7 describes the use of synchronization with Threading Building Blocks.

Table 2-1. Design spaces and Threading Building Blocks

Design space	Key	Your job or not?
Finding concurrency	Think Parallel	Your job
Algorithm structures	Algorithms from Chapters 3 and 4	You express yourself using the algorithm templates; the rest is taken care of for you
Supporting structures	Algorithms from Chapters 3 and 4	Threading Building Blocks
Implementation mechanisms	Write code to avoid the need for explicit synchronization, or use mutual exclusion from Chapter 7	Threading Building Blocks does most of the work for you

Our envelope-stuffing example from Figure 2-8 can be translated into code by addressing these four design spaces. For the *finding concurrency* design space, the concurrency is both data parallelism (the materials: envelopes, stamps, papers) and task parallelism (the jobs of folding, stuffing, etc.). For the *algorithm structures* design space, we chose a pipeline. With Threading Building Blocks, we can use `tbb::pipeline`. Because the synchronization is all implicit, there is no need for locks. Therefore, Threading Building Blocks handles the last two design spaces—*supporting structures* and *implementation mechanisms*—for us.

Intuition

After reading this chapter, you should be able to explain Thinking Parallel in terms of decomposition, scaling, correctness, abstraction, and patterns.

Once you understand these five concepts, and you can juggle them in your head when considering a programming task, you are Thinking Parallel. You will be developing an intuition about parallelism that will serve you well. Already, programmers seek to develop a sense of when a problem should use a parser, call on a sort algorithm, involve a database, use object-oriented programming, and so on. We look for patterns, think about decomposition, understand abstractions, and anticipate what approaches will be easier to debug. Parallelism is no different in this respect.

Developing an intuition to Think Parallel requires nothing more than understanding this chapter and trying it on a few projects. Intel Threading Building Blocks helps with each type of thinking presented in the chapter. Combined with Think Parallel intuition, Threading Building Blocks simply helps with parallel programming.

C++/Programming Languages

O'REILLY®

Intel Threading Building Blocks

"The Age of Serial Computing is over.... Not only does this book offer an excellent introduction to the library, it furnishes novices and experts alike with a clear and accessible discussion of the complexities of concurrency."
—Charles E. Leiserson, MIT Computer Science and Artificial Intelligence Laboratory

"We used to say make it right, then make it fast. We can't do that anymore. TBB lets us design for correctness and speed up front for Maya. This book shows you how to extract the most benefit from using TBB in your code."
—Martin Watt, Senior Software Engineer, Autodesk

This guide explains how you can maximize the benefits of multi-core processors by using Intel Threading Building Blocks (TBB), a portable C++ library that works on Windows, Linux, Macintosh, and Unix systems. Multi-core chips offer a dramatic boost in speed and responsiveness and opportunities for multiprocessing on ordinary desktop computers. But they also present a challenge: more than ever, multithreading is a requirement for good performance. With this book, you'll learn how to use TBB effectively for parallel programming—without having to be a threading expert.

Written by James Reinders, Chief Evangelist of Intel Software Products, and based on the experience of Intel's developers and customers, this book explains the key tasks in multithreading and how to accomplish them with TBB in a portable and robust manner. With plenty of examples and full reference material, this book lays out common patterns of use, reveals the gotchas in TBB, and gives important guidelines for choosing among alternatives in order to get the best performance. This book covers:

- Easy and effective ways to exploit the parallelism of multi-core systems
- The key issues in writing parallel programs
- Common patterns in multithreading
- Thread-safe containers for efficient processing
- Task scheduling
- Memory management in a threaded environment

Intel Threading Building Blocks demonstrates how TBB enables you to specify parallelism in C++ far more conveniently than using raw threads while improving performance, portability, and scalability. You don't need experience with parallel programming or multi-core processors to get started with TBB. Any C++ programmer who wants to write applications to run on multi-core systems will benefit from this book.

www.oreilly.com

US $34.99 **CAN $41.99**
ISBN-10: 0-596-51480-8
ISBN-13: 978-0-596-51480-8

53499
9 780596 514808

Structured Parallel Programming

Patterns for Efficient Computation

Michael McCool, Arch D. Robison, James Reinders

MK
MORGAN KAUFMANN

CHAPTER

Introduction 1

All computers are now parallel. Specifically, all modern computers support parallelism in hardware through at least one parallel feature, including vector instructions, multithreaded cores, **multicore processors**, multiple processors, graphics engines, and parallel co-processors. This statement does not apply only to supercomputers. Even the smallest modern computers, such as phones, support many of these features. It is also necessary to use explicit parallel programming to get the most out of such computers. Automatic approaches that attempt to parallelize serial code simply cannot deal with the fundamental shifts in algorithm structure required for effective parallelization.

Since parallel programming is no longer a special topic applicable to only select computers, this book is written with a simple premise: Parallel programming *is* programming. The evolution of computers has made parallel programming mainstream. Recent advances in the implementation of efficient parallel programs need to be applied to mainstream applications.

We explain how to design and implement efficient, reliable, and maintainable programs, in C and C++, that scale performance for all computers. We build on skills you already have, but without assuming prior knowledge of parallelism. Computer architecture issues are introduced where their impact must be understood in order to design an efficient program. However, we remain consistently focused on programming and the programmer's perspective, not on the hardware. This book is for programmers, not computer architects.

We approach the problem of practical parallel programming through a combination of patterns and examples. **Patterns** are, for our purposes in this book, valuable algorithmic structures that are commonly seen in efficient parallel programs. The kinds of patterns we are interested in are also called "algorithm skeletons" since they are often used as fundamental organizational principles for algorithms. The patterns we will discuss are expressions of the "best known solutions" used in effective and efficient parallel applications. We will discuss patterns both "from the outside," as abstractions, and "from the inside," when we discuss efficient implementation strategies. Patterns also provide a vocabulary to design new efficient parallel algorithms and to communicate these designs to others. We also include many examples, since examples show how these patterns are used in practice. For each example, we provide working code that solves a specific, practical problem.

Higher level programming models are used for examples rather than raw threading interfaces and vector intrinsics. The task of programming (formerly known as "parallel programming") is presented in a manner that focuses on capturing algorithmic intent. In particular, we show examples that are *appropriately* freed of unnecessary distortions to map algorithms to particular hardware. By focusing on the most important factors for performance and expressing those using models with low-overhead implementations, this book's approach to programming can achieve efficiency and scalability on a range of hardware.

The goal of a programmer in a modern computing environment is not just to take advantage of processors with two or four cores. Instead, it must be to write **scalable** applications that can take advantage of any amount of parallel hardware: all four cores on a quad-core processor, all eight cores on octo-core processors, thirty-two cores in a multiprocessor machine, more than fifty cores on new many-core processors, and beyond. As we will see, the quest for scaling requires attention to many factors, including the minimization of data movement, serial bottlenecks (including locking), and other forms of overhead. Patterns can help with this, but ultimately it is up to the diligence and intelligence of the software developer to produce a good algorithm design.

The rest of this chapter first discusses why it is necessary to "Think Parallel" and presents recent hardware trends that have led to the need for explicit parallel programming. The chapter then discusses the structured, pattern-based approach to programming used throughout the book. An introduction to the programming models used for examples and some discussion of the conventions and organization of this book conclude the chapter.

1.1 THINK PARALLEL

Parallelism is an intuitive and common human experience. Everyone reading this book would expect parallel checkout lanes in a grocery store when the number of customers wishing to buy groceries is sufficiently large. Few of us would attempt construction of a major building alone. Programmers naturally accept the concept of parallel work via a group of workers, often with specializations.

Serialization is the act of putting some set of operations into a specific order. Decades ago, computer architects started designing computers using serial machine languages to simplify the programming interface. **Serial semantics** were used even though the hardware was naturally parallel, leading to something we will call the **serial illusion**: a mental model of the computer as a machine that executes operations sequentially. This illusion has been successfully maintained over decades by computer architects, even though processors have become more and more parallel internally. The problem with the serial illusion, though, is that programmers came to depend on it too much.

Current programming practice, theory, languages, tools, data structures, and even most algorithms focus almost exclusively on serial programs and assume that operations are serialized. Serialization has been woven into the very fabric of the tools, models, and even concepts all programmers use. However, frequently serialization is actually unnecessary, and in fact is a poor match to intrinsically parallel computer hardware. Serialization is a learned skill that has been over-learned.

Up until the recent past, serialization was not a substantial problem. Mainstream computer architectures even in 2002 did not significantly penalize programmers for overconstraining algorithms with serialization. But now—they do. Unparallelized applications leave significant performance on the table for current processors. Furthermore, such serial applications will not improve in performance over time. Efficiently parallelized applications, in contrast, will make good use of current processors and should be able to scale automatically to even better performance on future processors. Over time, this will lead to large and decisive differences in performance.

Serialization has its benefits. It is simple to reason about. You can read a piece of serial code from top to bottom and understand the temporal order of operations from the structure of the source code. It helps that modern programming languages have evolved to use structured control flow to emphasize this aspect of serial semantics. Unless you intentionally inject randomness, serial programs also always do the same operations in the same order, so they are naturally **deterministic**. This means they give the same answer every time you run them with the same inputs. Determinism is useful for debugging, verification, and testing. However, deterministic behavior is not a natural characteristic of parallel programs. Generally speaking, the timing of task execution in parallel programs, in particular the relative timing, is often non-deterministic. To the extent that timing affects the computation, parallel programs can easily become non-deterministic.

Given that parallelism is necessary for performance, it would be useful to find an effective approach to parallel programming that retains as many of the benefits of serialization as possible, yet is also similar to existing practice.

In this book, we propose the use of structured patterns of parallelism. These are akin to the patterns of structured control flow used in serial programming. Just as structured control flow replaced the use of `goto` in most programs, these patterns have the potential to replace low-level and architecture-specific parallel mechanisms such as threads and vector intrinsics. An introduction to the pattern concept and a summary of the parallel patterns presented in this book are provided in Section 1.4. Patterns provide structure but have an additional benefit: Many of these patterns avoid non-determinism, with a few easily visible exceptions where it is unavoidable or necessary for performance. We carefully discuss when and where non-determinism can occur and how to avoid it when necessary.

Even though we want to eliminate unnecessary serialization leading to poor performance, current programming tools still have many **serial traps** built into them. Serial traps are constructs that make, often unnecessary, serial assumptions. Serial traps can also exist in the design of algorithms and in the abstractions used to estimate complexity and performance. As we proceed through this book, starting in Section 1.3.3, we will describe several of these serial traps and how to avoid them. However, serial semantics are still useful and should not be discarded in a rush to eliminate serial traps. As you will see, several of the programming models to be discussed are designed around generalizations of the semantics of serial programming models in useful directions. In particular, parallel programming models often try to provide equivalent behavior to a particular serial ordering in their parallel constructs, and many of the patterns we will discuss have serial equivalents. Using these models and patterns makes it easier to reason about and debug parallel programs, since then at least some of the nice properties of serial semantics can be retained.

Still, effective programming of modern computers demands that we regain the ability to "Think Parallel." Efficient programming will not come when parallelism is an afterthought. Fortunately, we can get most of "Think Parallel" by doing two things: (1) learning to recognize serial traps, some of which we examine throughout the remainder of this section, and (2) programming in terms of parallel patterns that capture best practices and using efficient implementations of these patterns.

Perhaps the most difficult part of learning to program in parallel is recognizing and avoiding serial traps—assumptions of serial ordering. These assumptions are so commonplace that often their existence goes unnoticed. Common programming idioms unnecessarily overconstrain execution order, making parallel execution difficult. Because serialization had little effect in a serial world, serial assumptions went unexamined for decades and many were even designed into our programming languages and tools.

We can motivate the **map** pattern (see Chapter 4) and illustrate the shift in thinking from serialized coding styles to parallel by a simple but real example.

For example, searching content on the World Wide Web for a specific phrase could be looked at as a serial problem or a parallel problem. A simplisitic approach would be to code such a search as follows:

```
for (i = 0; i < number_web_sites; ++i) {
   search(searchphrase, website[i]);
}
```

This uses a loop construct, which is used in serial programming as an idiom to "do something with a number of objects." However, what it actually means is "do something with a number of objects *one after the other*."

Searching the web as a parallel problem requires thinking more like

```
parallel_for (i = 0; i < number_web_sites; ++i) {
   search(searchphrase, website[i]);
}
```

Here the intent is the same—"do something with a number of objects"—but the constraint that these operations are done one after the other has been removed. Instead, they may be done simultaneously.

However, the serial semantics of the original `for` loop allows one search to leave information for the next search to use if the programmer so chooses. Such temptation and opportunity are absent in the `parallel_for` which requires each invocation of the search algorithm to be independent of other searches. That fundamental shift in thinking, to using parallel patterns when appropriate, is critical to harness the power of modern computers. Here, the `parallel_for` implements the map pattern (described in Chapter 4). In fact, different uses of iteration (looping) with different kinds of dependencies between iterations correspond to different parallel patterns. To parallelize serial programs written using iteration constructs you need to recognize these idioms and convert them to the appropriate parallel structure. Even better would be to design programs using the parallel structures in the first place.

In summary, if you do not already approach every computer problem with parallelism in your thoughts, we hope this book will be the start of a new way of thinking. Consider ways in which you may be unnecessarily serializing computations. Start thinking about how to organize work to expose parallelism and eliminate unnecessary ordering constraints, and begin to "Think Parallel."

1.2 PERFORMANCE

Perhaps the most insidious serial trap is our affection for discussing algorithm performance with all attention focused on the minimization of the total amount of computational work. There are two problems with this. First of all, computation may not be the bottleneck. Frequently, access to memory or (equivalently) *communication* may constrain performance. Second, the potential for scaling performance on a parallel computer is constrained by the algorithm's **span**. The span is the time it takes to perform the longest chain of tasks that must be performed sequentially. Such a chain is known as a critical path, and, because it is inherently sequential, it cannot be sped up with parallelism, no matter how many parallel processors you have. The span is a crucial concept which will be used throughout the book. Frequently, getting **improved performance** requires finding an alternative way to solve a problem that shortens the span.

This book focuses on the **shared memory** machine model, in which all parts of application have access to the same shared memory address space. This machine model makes communication implicit: It happens automatically when one worker writes a value and another one reads it. Shared memory is convenient but can hide communication and can also lead to unintended communication. Unfortunately, communication is not free, nor is its cost uniform. The cost in time and energy of communication varies depending upon the location of the worker. The cost is minimal for lanes of a vector unit (a few instructions), relatively low for hardware threads on the same core, more for those sharing an on-chip cache memory, and yet higher for those in different sockets.

Fortunately, there is a relatively simple abstraction, called **locality**, that captures most of these cost differences. The locality model asserts that memory accesses close together in time and space (and communication between processing units that are close to each other in space) are cheaper than those that are far apart. This is not completely true—there are exceptions, and cost is non-linear with respect to locality—but it is better than assuming that all memory accesses are uniform in cost. Several of the data access patterns in this book are used to improve locality. We also describe several pitfalls in memory usage that can hurt performance, especially in a parallel context.

The concept of span was previously mentioned. The span is the critical path or, equivalently, the longest chain of operations. To achieve scaling, minimizing an algorithm's span becomes critical. Unsurprisingly, parallel programming is simplest when the tasks to be done are completely independent. In such cases, the span is just the longest task and communication is usually negligible (not zero, because we still have to check that all tasks are done). Parallel programming is much more challenging when tasks are not independent, because that requires communication between tasks, and the span becomes less obvious.

Span determines a limit on how fast a parallel algorithm can run even given an infinite number of cores and infinitely fast communication. As a simple example, if you make pizza from scratch, having several cooks can speed up the process. Instead of preparing dough, sauce, and topping one at a time (serially), multiple cooks can help by mixing the dough and preparing the toppings in parallel. But the crust for any given pizza takes a certain amount of time to bake. That time contributes to the span of making a single pizza. An infinite number of cooks cannot reduce the cooking time, even if they can prepare the pizza faster and faster before baking. If you have heard of **Amdahl's Law** giving an upper bound on scalability, this may sound familiar. However, the concept of span is more precise, and gives tighter bounds on achievable scaling. We will actually show that Amdahl was both an optimist and a pessimist. Amdahl's Law is a relatively loose upper bound on scaling. The use of the **work-span** model provides a tighter bound and so is more realistic, showing that Amdahl was an optimist. On the other hand, the scaling situation is usually much less pessimistic if the size of the problem is allowed to grow with the number of cores.

When designing a parallel algorithm, it is actually important to pay attention to three things:

- The total amount of computational work.
- The span (the critical path).
- The total amount of communication (including that implicit in sharing memory).

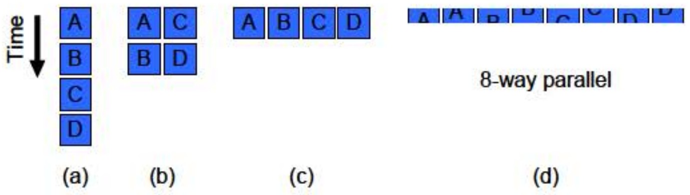

FIGURE 1.1

Independent software tasks can be run in parallel on multiple workers. In theory, this can give a linear speedup. In reality, this is a gross oversimplification. It may not be possible to uniformly subdivide an application into independent tasks, and there may be additional overhead and communication resulting from the subdivision.

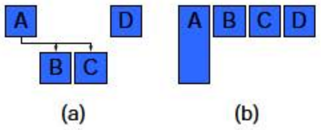

FIGURE 1.2

Tasks running in parallel: some more complex situations. (a) Tasks can be arranged to run in parallel as long as dependencies are honored. (b) Tasks may take different amounts of time to execute. Both of these issues can increase the span and reduce scalability.

In order for a program to scale, span and communication limitations are as important to understand and minimize as the total computational work.

A few examples are probably helpful at this point. In Figure 1.1a, a serial program with no parallelism simply performs tasks A, B, C, and D in sequence. As a convention, the passage of time will be shown in our diagrams as going from top to bottom. We highlight this here with an arrow showing the progress of time, but will generally just assume this convention elsewhere in the book.

A system with two parallel workers might divide up work so one worker performs tasks A and B and the other performs tasks C and D, as shown in Figure 1.1b. Likewise, a four-way system might perform tasks A, B, C, and D, each using separate resources as shown in Figure 1.1c. Maybe you could even contemplate subdividing the tasks further as shown in Figure 1.1d for eight workers. However, this simple model hides many challenges. What if the tasks depend on each other? What if some tasks take longer to execute than others? What if subdividing the tasks into subtasks requires extra work? What if some tasks cannot be subdivided? What about the costs for communicating between tasks?

If the tasks were not independent we might have to draw something like Figure 1.2a. This illustration shows that tasks A and D are independent of each other, but that tasks B and C have a dependency on A completing first. Arrows such as these will be used to show dependencies in this book, whether they are data or control dependencies. If the individual tasks cannot be subdivided further, then the running time of the program will be at least the sum of the running time of tasks A and B or the sum of the running time of tasks A and C, whichever is longer. This is the span of this parallel algorithm. Adding more workers cannot make the program go faster than the time it takes to execute the span.

In most of this book, the illustrations usually show tasks as having equal size. We do not mean to imply this is true; we do it only for ease of illustration. Considering again the example in Figure 1.1c, even if the tasks are completely independent, suppose task A takes longer to run than the others. Then the illustration might look like Figure 1.2b. Task A alone now determines the span.

We have not yet considered limitations due to communication. Suppose the tasks in a parallel program all compute a partial result and they need to be combined to produce a final result. Suppose that this combination is simple, such as a summation. In general, even such a simple form of communication, which is called a **reduction**, will have a span that is logarithmic in the number of workers involved.

Effectively addressing the challenges of decomposing computation and managing communications are essential to efficient parallel programming. Everything that is unique to parallel programming will be related to one of these two concepts. Effective parallel programming requires effective management of the distribution of work and control of the communication required. Patterns make it easier to reason about both of these. Efficient programming models that support these patterns, that allow their efficient implementation, are also essential.

For example, one such implementation issue is **load balancing**, the problem of ensuring that all processors are doing their fair share of the work. A load imbalance can result in many processors idling while others are working, which is obviously an inefficient use of resources. The primary programming models used in this book, Cilk Plus and TBB, both include efficient work-stealing schedulers to efficiently and automatically balance the load. Basically, when workers run out of things to do, they actively find new work, without relying on a central manager. This decentralized approach is much more scalable than the use of a centralized work-list. These programming models also provide mechanisms to subdivide work to an appropriate **granularity** on demand, so that tasks can be decomposed when more workers are available.

1.3 MOTIVATION: PERVASIVE PARALLELISM

Parallel computers have been around for a long time, but several recent trends have led to increased parallelism at the level of individual, mainstream personal computers. This section discusses these trends. This section also discusses why taking advantage of parallel hardware now generally requires explicit parallel programming.

1.3.1 Hardware Trends Encouraging Parallelism

In 1965, Gordon Moore observed that the number of transistors that could be integrated on silicon chips were doubling about every 2 years, an observation that has become known as **Moore's Law**. Consider Figure 1.3, which shows a plot of transistor counts for Intel microprocessors. Two rough data points at the extremes of this chart are on the order of 1000 (10^3) transistors in 1971 and about 1000 million (10^9) transistors in 2011. This gives an average slope of 6 orders of magnitude over 40 years, a rate of 0.15 orders of magnitude every year. This is actually about 1.41× per year, or 1.995× every 2 years. The data shows that Moore's original prediction of 2× per year has been amazingly accurate. While we only give data for Intel processors, processors from other vendors have shown similar trends.

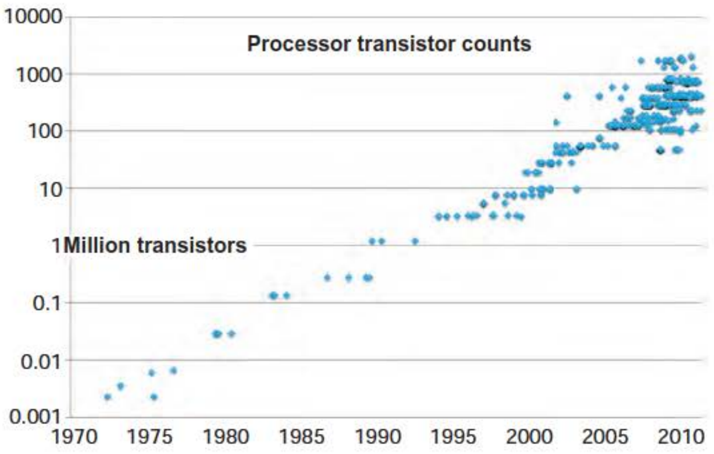

FIGURE 1.3

Moore's Law, which states roughly that the number of transistors that can be integrated on a chip will double about every 2 years, continues to this day (log scale). The straight line on this graph, which is on a logarithmic scale, demonstrates exponential growth in the total number of transistors in a processor from 1970 to the present. In more recent times, with the advent of multicore processors, different versions of processors with different cache sizes and core counts have led to a greater diversity in processor sizes in terms of transistor counts.

This exponential growth has created opportunities for more and more complex designs for microprocessors. Until 2004, there was also a rise in the switching speed of transistors, which translated into an increase in the performance of microprocessors through a steady rise in the rate at which their circuits could be clocked. Actually, this rise in clock rate was also partially due to architectural changes such as instruction pipelining, which is one way to automatically take advantage of instruction-level parallelism. An increase in clock rate, if the instruction set remains the same (as has mostly been the case for the Intel architecture), translates roughly into an increase in the rate at which instructions are completed and therefore an increase in computational performance. This increase is shown in Figure 1.4. Actually, many of the increases in processor complexity have also been to increase performance, even on single-core processors, so the actual increase in performance has been greater than this.

From 1973 to 2003, clock rates increased by three orders of magnitude (1000×), from about 1 MHz in 1973 to 1 GHz in 2003. However, as is clear from this graph clock rates have now ceased to grow and are now generally in the 3 GHz range. In 2005, three factors converged to limit the growth in performance of single cores and shift new processor designs to the use of multiple cores. These are known as the "three walls":

Power wall: Unacceptable growth in power usage with clock rate.
Instruction-level parallelism (ILP) wall: Limits to available low-level parallelism.
Memory wall: A growing discrepancy of processor speeds relative to memory speeds.

can be decomposed. Ten stages is about the maximum useful limit, although there have been processors with 31 stages [DF90]. It is even possible for a processor to issue instructions speculatively, in order to increase parallelism. However, since speculation results in wasted computation it can be expensive from a power point of view. Modern processors do online program analysis, such as maintaining branch history tables to try to increase the performance of speculative techniques such as branch prediction and prefetching, which can be very effective, but they themselves take space and power, and programs are by nature not completely predictable. In the end, ILP can only deliver constant factors of speedup and cannot deliver continuously scaling performance over time.

Programming has long been done primarily as if computers were serial machines. Meanwhile, computer architects (and compiler writers) worked diligently to find ways to automatically extract parallelism, via ILP, from their code. For 40 years, it was possible to maintain this illusion of a serial programming model and write reasonably efficient programs while largely ignoring the true parallel nature of hardware. However, the point of decreasing returns has been passed with ILP techniques, and most computer architects believe that these techniques have reached their limit. The ILP wall reflects the fact that the automatically extractable low-level parallelism has already been used up.

The memory wall results because off-chip memory rates have not grown as fast as on-chip computation rates. This is due to several factors, including power and the number of pins that can be easily incorporated into an integrated package. Despite recent advances, such as double-data-rate (DDR) signaling, off-chip *communication* is still relatively slow and power-hungry. Many of the transistors used in today's processors are for cache, a form of on-chip memory that can help with this problem. However, the performance of many applications is fundamentally bounded by memory performance, not compute performance. Many programmers have been able to ignore this due to the effectiveness of large caches for serial processors. However, for parallel processors, interprocessor communication is also bounded by the memory wall, and this can severely limit scalability. Actually, there are two problems with memory (and communication): **latency** and **bandwidth**. Bandwidth (overall data rate) can still be scaled in several ways, such as optical interconnections, but latency (the time between when a request is submitted and when it is satisfied) is subject to fundamental limits, such as the speed of light. Fortunately, as discussed later in Section 2.5.9, latency can be hidden—given sufficient additional parallelism, above and beyond that required to satisfy multiple computational units. So the memory wall has two effects: Algorithms need to be structured to avoid memory access and communication as much as possible, and fundamental limits on latency create even more requirements for parallelism.

In summary, in order to achieve increasing performance over time for each new processor generation, you cannot depend on rising clock rates, due to the power wall. You also cannot depend on automatic mechanisms to find (more) parallelism in naïve serial code, due to the ILP wall. To achieve higher performance, you now *have* to write explicitly parallel programs. And finally, when you write these parallel programs, the memory wall means that you also have to seriously consider communication and memory access costs and may even have to use additional parallelism to hide latency.

Instead of using the growing number of transistors predicted by Moore's Law for ways to maintain the ''serial processor illusion,'' architects of modern processor designs now provide multiple mechanisms for explicit parallelism. However, you must use them, and use them well, in order to achieve performance that will continue to scale over time.

The resulting trend in hardware is clear: More and more parallelism at a hardware level will become available for any application that is written to utilize it. However, unlike rising clock rates, non-parallelized application performance will not change without active changes in programming. The "free lunch" [Sut05] of automatically faster serial applications through faster microprocessors has ended. The new "free lunch" requires scalable parallel programming. The good news is that if you design a program for scalable parallelism, it will continue to scale as processors with more parallelism become available.

1.3.2 Observed Historical Trends in Parallelism

Parallelism in hardware has been present since the earliest computers and reached a great deal of sophistication in mainframe and vector supercomputers by the late 1980s. However, for mainstream computation, miniaturization using integrated circuits started with designs that were largely devoid of hardware parallelism in the 1970s. Microprocessors emerged first using simple single-threaded designs that fit into an initially very limited transistor budget. In 1971, the Intel 4004 4-bit microprocessor was introduced, designed to be used in an electronic calculator. It used only 2,300 transistors in its design. The most recent Intel processors have enough transistors for well over a million Intel 4004 microprocessors. The Intel Xeon E7-8870 processor uses 2.6×10^9 transistors, and the upcoming Intel MIC architecture co-processor, known as Knights Corner, is expected to roughly double that. While a processor with a few million cores is unlikely in the near future, this gives you an idea of the potential.

Hardware is naturally parallel, since each transistor can switch independently. As transistor counts have been growing in accordance with Moore's Law, as shown in Figure 1.3, hardware parallelism, both implicit and explicit, gradually also appeared in microprocessors in many forms. Growth in word sizes, superscalar capabilities, vector (SIMD) instructions, out-of-order execution, multithreading (both on individual cores and on multiple cores), deep pipelines, parallel integer and floating point arithmetic units, virtual memory controllers, memory prefetching, page table walking, caches, memory access controllers, and graphics processing units are all examples of using additional transistors for parallel capabilities.

Some variability in the number of transistors used for a processor can be seen in Figure 1.3, especially in recent years. Before multicore processors, different cache sizes were by far the driving factor in this variability. Today, cache size, number of cores, and optional core features (such as vector units) allow processors with a range of capabilities to be produced. This is an additional factor that we must take into account when writing a program: Even at a single point in time, a program may need to run on processors with different numbers of cores, different vector instruction sets and vector widths, different cache sizes, and possibly different instruction latencies.

The extent to which software needed to change for each kind of additional hardware mechanism using parallelism has varied a great deal. Automatic mechanisms requiring the least software change, such as instruction-level parallelism (ILP), were generally introduced first. This worked well until several issues converged to force a shift to explicit rather than implicit mechanisms in the multicore era. The most significant of these issues was power. Figure 1.5 shows a graph of total power consumption over time. After decades of steady increase in power consumption, the so-called *power wall* was hit about 2004. Above around 130W, air cooling is no longer practical. Arresting power growth required that clock rates stop climbing. From this chart we can see that modern processors now span a large range of power consumption, with the availability of lower power parts driven by the growth of mobile and embedded computing.

The resulting trend toward explicit parallelism mechanisms is obvious looking at Figure 1.6, which plots the sudden rise in the number of hardware threads[1] after 2004. This date aligns with the halt in the

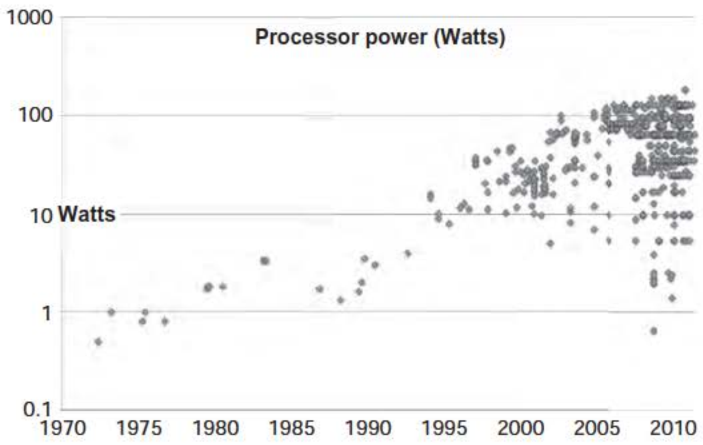

FIGURE 1.5

Graph of processor total power consumption (log scale). The maximum power consumption of processors saw steady growth for nearly two decades before the multicore era. The inability to dissipate heat with air cooling not only brought this growth to a halt but increased interest in reduced power consumption, greater efficiencies, and mobile operation created more options at lower power as well.

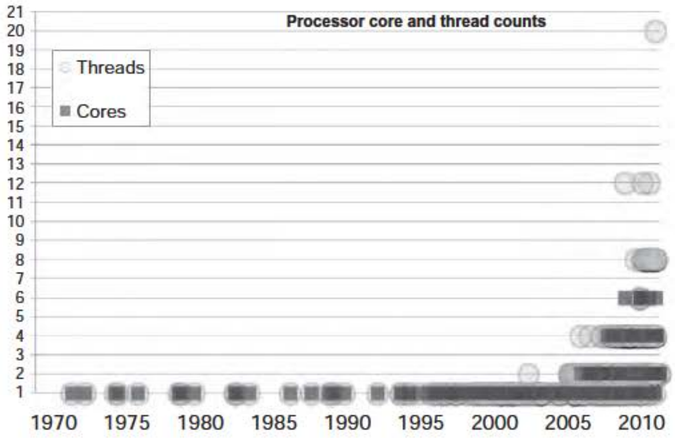

FIGURE 1.6

The number of cores and hardware threads per processor was one until around 2004, when growth in hardware threads emerged as the trend instead of growth in clock rate.

[1]It is common to refer to hardware parallelism as processor cores and to stress multicore. But it is more precise to speak of hardware threads, since some cores can execute more than one thread at a time. We show both in the graph.

growth in clock rate. The power problem was arrested by adding more cores and more threads in each core rather than increasing the clock rate. This ushered in the multicore era, but using multiple hardware threads requires more software changes than prior changes. During this time vector instructions were added as well, and these provide an additional, multiplicative form of explicit parallelism. **Vector parallelism** can be seen as an extension of data width parallelism, since both are related to the width of hardware registers and the amount of data that can be processed with a single instruction. A measure of the growth of data width parallelism is shown in Figure 1.7. While data width parallelism growth predates the halt in the growth of clock rates, the forces driving multicore parallelism growth are also adding motivation to increase data width. While some automatic **parallelization** (including **vectorization**) is possible, it has not been universally successful. Explicit parallel programming is generally needed to fully exploit these two forms of hardware parallelism capabilities.

Additional hardware parallelism will continue to be motivated by Moore's Law coupled with power constraints. This will lead to processor designs that are increasingly complex and diverse. Proper abstraction of parallel programming methods is necessary to be able to deal with this diversity and to deal with the fact that Moore's Law continues unabated, so the maximum number of cores (and the diversity of processors) will continue to increase.

Counts of the number of hardware threads, vector widths, and clock rates are only indirect measures of performance. To get a more accurate picture of how performance has increased over time, looking at

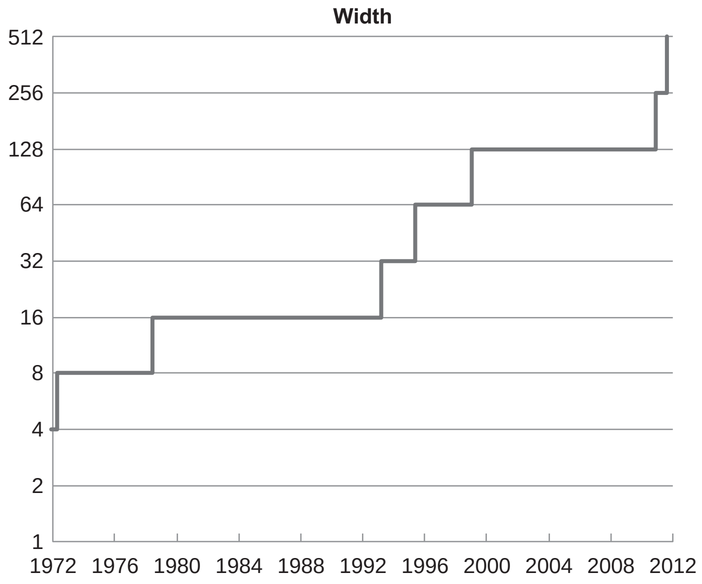

FIGURE 1.7

Growth in data processing widths (log scale), measured as the number of bits in registers over time. At first the width of scalar elements grew, but now the number of elements in a register is growing with the addition of vector (SIMD) instructions that can specify the processing of multiple scalar elements at once.

benchmarks can be helpful. Unfortunately, long-term trend analysis using benchmarks is difficult due to changes in the benchmarks themselves over time.

We chose the industry standard CPU2006 SPEC benchmarks. Unfortunately, these are exclusively from the multicore era as they only provide data from 2006 [Sub06]. In preparing the graphs in this section of our book, we also choose to show only data related to Intel processors. Considering only one vendor avoids a certain blurring effect that occurs when data from multiple vendors is included. Similar trends are observable for processors from other vendors, but the trends are clearer when looking at data from a single vendor.

Some discussion of the nature of the CPU2006 benchmarks is important so the results can be properly understood. First, these benchmarks are not explicitly parallelized, although autoparallelization is allowed. Autoparallelization must be reported, however, and may include the use of already-parallelized libraries. It is however not permitted to change the source code of these benchmarks, which prohibits the use of new parallel programming models. In fact, even standardized OpenMP directives, which would allow explicit parallelization, must be explicitly disabled by the SPEC run rules. There are SPEC benchmarks that primarily stress floating point performance and other benchmarks that primarily stress integer and control flow performance. The FP and INT designations indicate the floating-point and integer subsets. INT benchmarks usually also include more complex control flow. The "rate" designations indicate the use of multiple copies of the benchmarks on computers with multiple hardware threads in order to measure throughput. These "rate" (or throughput) results give some idea of the potential for speedup from parallelism, but because the benchmark instances are completely independent these measurements are optimistic.

Figures 1.8, 1.9, and 1.10 show SPEC2006 benchmark results that demonstrate what has happened to processor performance during the multicore era (since 2006). Figure 1.8 shows that performance per Watt has improved considerably for entire processors as the core count has grown. Furthermore, on multiprocessor computers with larger numbers of cores, Figure 1.9 shows that **throughput** (the total performance of multiple independent applications) has continued to scale to considerably higher performance. However, Figure 1.10 shows that the performance of individual benchmarks has remained nearly flat, even though autoparallelization is allowed by the SPEC process. The inescapable conclusion is that, while overall system performance is increasing, increased performance of single applications requires *explicit* parallelism in software.

1.3.3 Need for Explicit Parallel Programming

Why can't parallelization be done automatically? Sometimes it can be, but there are many difficulties with automatically parallelizing code that was originally written under the assumption of serial execution, and in languages designed under that assumption.

We will call unnecessary assumptions deriving from the assumption of serial execution **serial traps**. The long-sustained serial illusion has caused numerous serial traps to become built into both our tools and ways of thinking. Many of these force serialization due to over-specification of the computation. It's not that programmers wanted to force serialization; it was simply assumed. Since it was convenient and there was no penalty at the time, serial assumptions have been incorporated into nearly everything. We will give several examples in this section. We call these assumptions "traps" because they cause modern systems to be unable to use parallelism even though the algorithm writer did not explicitly intend to forbid it.

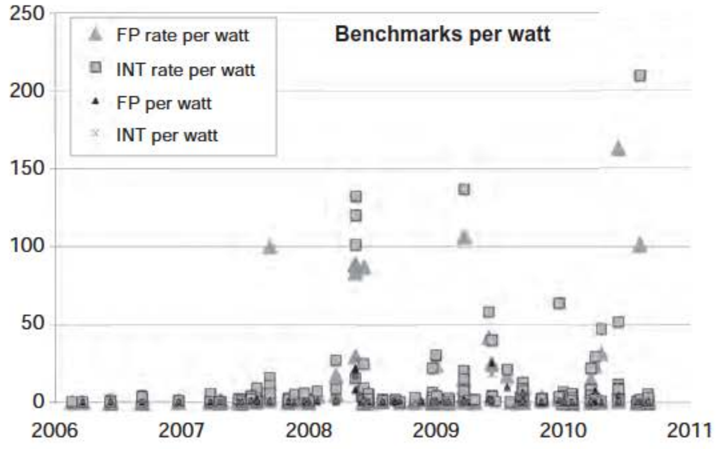

FIGURE 1.8

Performance per Watt using data derived from SPEC2006 benchmarks and processor (not system) power ratings from Intel corporation. The FP per Watt and INT per Watt give single benchmark performance. Autoparallelization is allowed but for the most part these benchmarks are not parallelized. The FP rate and INT rate per Watt results are based on running multiple copies of the benchmark on a single processor and are meant to measure throughput. The FP and INT results have not increased substantially over this time period, but the FP rate and INT rate results have. This highlights the fact that performance gains in the multicore era are dominated by throughput across cores, not from increased performance of a core.

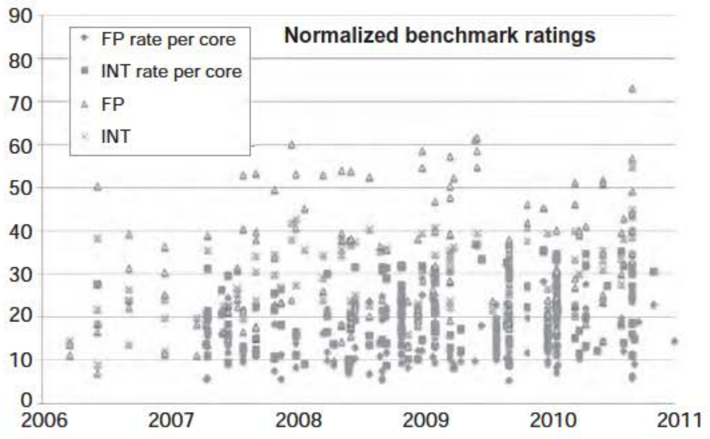

FIGURE 1.9

Performance in the multicore era, on a per hardware thread basis, does not show a strong and obvious trend as it did in the single-core megahertz era. Data derived from SPEC2006 benchmarks and processor (not system) power ratings, but with rate results divided by the number of parallel benchmark instances (hardware threads) used.

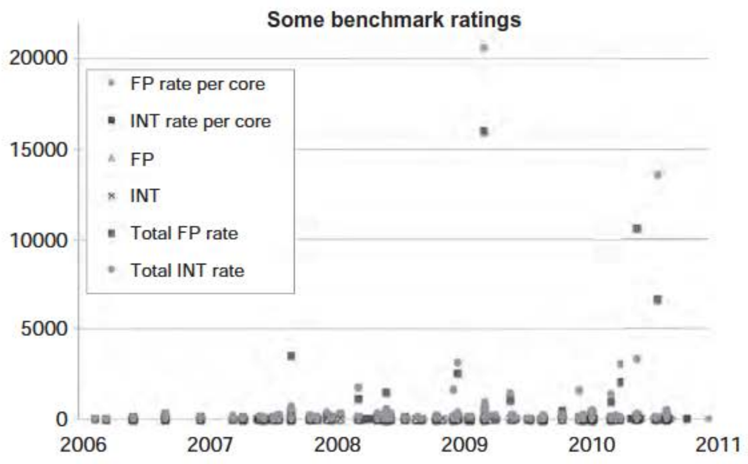

FIGURE 1.10

SPEC2006 performance on multiprocessor computers in the multicore era. Large machines can yield overall systems performance that dwarfs the per core performance numbers (note the two orders of magnitude shift in *Y*-axis scale vs. Figure 1.9). Data derived from SPEC benchmark archives.

Accidents of language design can make it difficult for compilers to discover parallelism or prove that it is safe to parallelize a region of code. Compilers *are* good at "packaging" parallelism they see even if it takes many detailed steps to do so. Compilers are not reliable at discovering parallelism opportunities. Frequently, the compiler cannot disprove some minor detail that (rarely) *might* be true that would make parallelism impossible. Then, to be safe, in such a situation it cannot parallelize.

Take, for example, the use of pointers. In C and C++, pointers allow the modification of any region of memory, at any time. This is very convenient and maps directly onto the underlying machine language mechanism (itself an abstraction of the hardware...) for memory access. With serial semantics, even with this freedom it is still clear what the state of memory will be at any time. With parallel hardware, this freedom becomes a nightmare. While great strides have been made in automatic pointer analysis, it is still difficult for a compiler in many situations to determine that the data needed for parallel execution will not be modified by some other part of the application at an inconvenient time, or that data references do not overlap in a way that would cause different orders of execution to produce different results.

Parallelism can also be hidden because serial control constructs, in particular loops, over-specify ordering. Listing 1.1 through Listing 1.7 show a few other examples of hiding parallelism that are common practice in programming languages that were not initially designed to allow explicit parallel programming. Parallel programming models often provide constructs that avoid some of these constraints. For concreteness, in this section we will show several solutions in Cilk Plus that remove these serial constraints and allow parallelism.

The straightforward C code in Listing 1.1 cannot be parallelized by a compiler in general because the arrays a, b, and c might partially overlap, as in Listing 1.2. The possibility of overlap adds a serial

```
void
addme(int n, double a[n], double b[n], double c[n]) {
   int i;
   for (i = 0; i < n; ++i)
      a[i] = b[i] + c[i];
}
```

LISTING 1.1

Add two vectors in C, with implied serial ordering.

```
double a[10];
a[0] = 1;
addme(9, a+1, a, a); // pointer arithmetic causing aliasing
```

LISTING 1.2

Overlapping (aliased) arguments in C. By calling the *serial* `addme` with overlapping arguments, this code fills `a` with powers of two. Such devious but legal usage is probably unintended by the author of `addme`, but the compiler does not know that.

```
void
addme(int n, double a[n], double b[n], double c[n]) {
   a[:] = b[:] + c[:];
}
```

LISTING 1.3

Add two vectors using Cilk Plus array notation.

constraint, even if the programmer never intended to exploit it. Parallelization requires reordering, but usually you want all the different possible orders to produce the same result.

A syntax that treats arrays as a whole, as shown in Listing 1.3, makes the parallelism accessible to the compiler by being explicit. This Cilk Plus array notation used here actually allows for simpler code than the loop shown in Listing 1.1, as well. However, use of this syntax also *requires* that the arrays not be partially overlapping (see Section B.8.5), unlike the code in Listing 1.1. This additional piece of information allows the compiler to parallelize the code.

Loops can specify different kinds of computations that must be parallelized in different ways. Consider Listing 1.4. This is a common way to sum the elements of an array in C.

Each loop iteration depends on the prior iteration, and thus the iterations cannot be done in parallel. However, if reordering floating-point addition is acceptable here, this loop *can* be both parallelized and vectorized, as explained in Section 5.1. But the compiler alone cannot tell whether the serial dependency was deliberate or just convenient. Listing 1.5 shows a way to convey parallel intent, both to the compiler and a human maintainer. It specifies a parallel loop and declares `mysum` in a way that

```
double summe(int n, double a[n]) {
   double mysum = 0;
   int i;
   for (i = 0; i < n; ++i)
      mysum += a[i];
   return mysum;
}
```

LISTING 1.4

An ordered sum creates a dependency in C.

```
double summe(int n, double a[n]) {
   sum_reducer<double> mysum (0);
   cilk_for (int i = 0; i < n; ++i)
      mysum += a[i];
  return mysum.get_value();
}
```

LISTING 1.5

A parallel sum, expressed as a reduction operation in Cilk Plus.

```
void callme() {
   foo();
   bar();
}
```

LISTING 1.6

Function calls with step-by-step ordering specified in C.

says that ordering the individual operations making up the sum is okay. This additional freedom allows the system to choose an order that gives the best performance.

As a final example, consider Listing 1.6, which executes `foo` and `bar` in exactly that order. Suppose that `foo` and `bar` are separately compiled library functions, and the compiler does not have access to their source code. Since `foo` *might* modify some global variable that `bar` might depend on, and the compiler cannot prove this is not the case, the compiler *has* to execute them in the order specified in the source code.

However, suppose you modify the code to explicitly state that `foo` and `bar` can be executed in parallel, as shown in Listing 1.7. Now the programmer has given the compiler permission to execute these functions in parallel. It does not mean the system *will* execute them in parallel, but it now has the option, if it would improve performance.

```
void callme() {
   cilk_spawn foo();
   bar();
}
```

LISTING 1.7

Function calls with no required ordering in Cilk Plus.

Later on we will discuss the difference between **mandatory parallelism** and **optional parallelism**. Mandatory parallelism forces the system to execute operations in parallel but may lead to poor performance—for example, in the case of a recursive program generating an exponential number of threads. Mandatory parallelism also does not allow for hierarchical composition of parallel software components, which has a similar problem as recursion. Instead, the Cilk Plus `cilk_spawn` notation simply identifies tasks that are *opportunities* for parallelism. It is up to the system to decide when, where, and whether to use that parallelism. Conversely, when you use this notation you should not assume that the two tasks are necessarily active simultaneously. Writing portable parallel code means writing code that can deal with any order of execution—including serial ordering.

Explicit parallel programming constructs allow algorithms to be expressed without specifying unintended and unnecessary serial constraints. Avoiding specifying ordering and other constraints when they are not required is fundamental. Explicit parallel constructs also provide additional information, such as declarations of independence of data and operations, so that the system implementing the programming model knows that it can safely execute the specified operations in parallel. However, the programmer now has to ensure that these additional constraints are met.

1.4 STRUCTURED PATTERN-BASED PROGRAMMING

History does not repeat itself, but it rhymes.

(attributed to Mark Twain)

In this book, we are taking a structured approach to parallel programming, based on patterns.

Patterns can be loosely defined as commonly recurring strategies for dealing with particular problems. Patterns have been used in architecture [Ale77], natural language learning [Kam05], object-oriented programming [GHJV95], and software architecture [BMR+96, SSRB00]. Others have also applied patterns specifically to parallel software design [MAB+02, MSM04, MMS05], as we do here. One notable effort is the OUR pattern language, an ongoing project to collaboratively define a set of parallel patterns [Par11].

We approach patterns as tools, and we emphasize patterns that have proven useful as tools. As such, the patterns we present codify practices and distill experience in a way that is reusable. In this book, we discuss several prerequisites for achieving parallel scalability, including good data locality and avoidance of overhead. Fortunately, many good strategies have been developed for achieving these objectives.

We will focus on **algorithm strategy patterns**, as opposed to the more general **design patterns** or system-specific **implementation patterns**. Design patterns emphasize high-level design processes.

These are important but rather abstract. Conversely, implementation patterns address low-level details that are often specific to a particular machine architecture, although occasionally we will discuss important low-level issues if they seriously impact performance. Algorithm strategy patterns lie in between these two extremes. They affect how your algorithms are organized, and so are also known as **algorithmic skeletons** [Col89, AD07].

Algorithm strategy patterns have two parts: semantics and implementation. The semantics describe how the pattern is used as a building block of an algorithm, and consists of a certain arrangement of tasks and data dependencies. The semantic view is an abstraction that intentionally hides some details, such as whether the tasks making up the pattern will actually run in parallel in a particular implementation. The semantic view of a pattern is used when an algorithm is designed. However, patterns also need to be implemented well on real machines. We will discuss several issues related to the implementation of patterns, including (for example) granularity control and good use of cache. The key point is that different implementation choices may lead to different performances, but *not* to different semantics. This separation makes it possible to reason about the high-level algorithm design and the low-level (and often machine-specific) details separately. This separation is not perfect; sometimes you will want to choose one pattern over another based on knowledge of differences in implementation. That's all right. Abstractions exist to simplify and structure programming, not to obscure important information.

Algorithm strategy patterns tend to map onto programming model features as well, and so are useful in understanding programming models. However, algorithm strategy patterns transcend particular languages or programming models. They do not have to map directly onto a programming language feature to be usable. Just as it is possible to use structured control flow in FORTRAN 66 by following conventions for disciplined use of `goto`, it is possible to employ the parallel patterns described in this book even in systems that do not directly support them. The patterns we present, summarized in Figure 1.11, will occur (or be usable) in almost any sufficiently powerful parallel programming model, and if used well should lead to well-organized and efficient programs with good scaling properties. Numerous examples in this book show these patterns in practice. Like the case with structured control flow in serial programming, structured parallel patterns simplify code and make it more understandable, leading to greater maintainability.

Three patterns deserve special mention: **nesting**, **map**, and **fork–join**. Nesting means that patterns can be hierarchically composed. This is important for modular programming. Nesting is extensively used in serial programming for **composability** and information hiding, but is a challenge to carry over into parallel programming. The key to implementing nested parallelism is to specify optional, not mandatory, parallelism. The map pattern divides a problem into a number of uniform parts and represents a regular parallelization. This is also known as **embarrassing parallelism**. The map pattern is worth using whenever possible since it allows for both efficient parallelization and efficient vectorization. The fork–join pattern recursively subdivides a problem into subparts and can be used for both regular and irregular parallelization. It is useful for implementing a **divide-and-conquer** strategy. These three patterns also emphasize that in order to achieve scalable parallelization we should focus on **data parallelism**: the subdivision of the problem into subproblems, with the number of subproblems able to grow with the overall problem size.

In summary, patterns provide a common vocabulary for discussing approaches to problem solving and allow reuse of best practices. Patterns transcend languages, programming models, and even computer architectures, and you can use patterns whether or not the programming system you are using explicitly supports a given pattern with a specific feature.

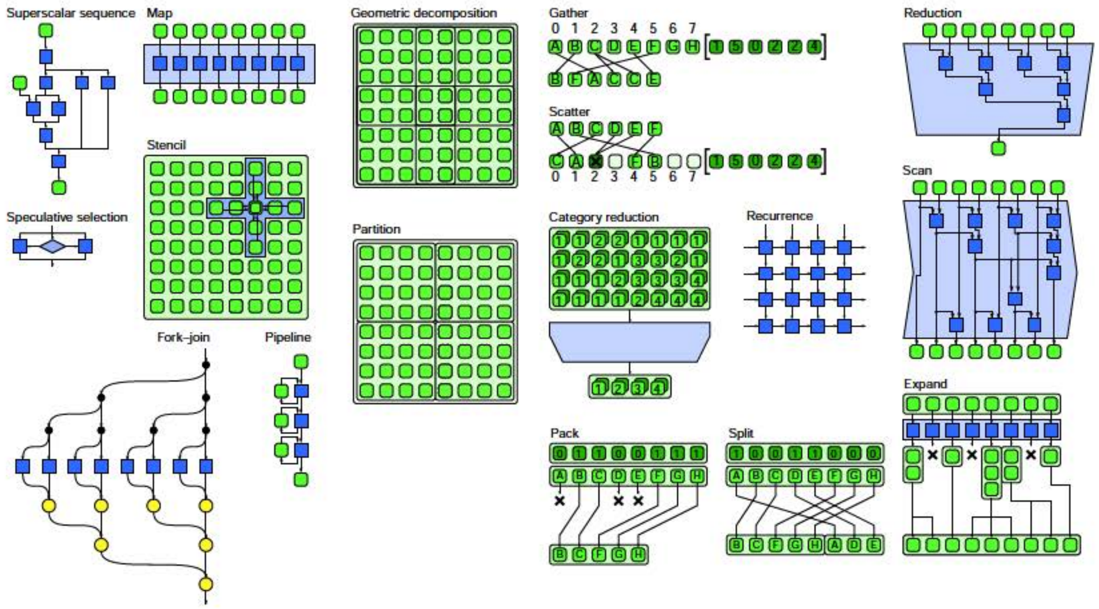

FIGURE 1.11

Overview of parallel patterns.

1.5 PARALLEL PROGRAMMING MODELS

We will discuss parallel programming models that can support a wide range of parallel programming needs. This section gives some basic background on the programming models used in this book. It will also discuss what makes a good programming model. Appendices B and C provide more information on the primary programming models used for examples in this book, TBB and Cilk Plus, as well as links to online resources.

1.5.1 Desired Properties

Unfortunately, none of the most popular programming languages in use today was designed for parallel programming. However, since a large amount of code has already been written in existing serial languages, practically speaking it is necessary to find an evolutionary path that extends existing programming practices and tools to support parallelism. Broadly speaking, while enabling dependable results, parallel programming models should have the following properties:

Performance: Achievable, scalable, predictable, and tunable. It should be possible to predictably achieve good performance and to scale that performance to larger systems.

Productivity: Expressive, composable, debuggable, and maintainable. Programming models should be complete and it should be possible to directly and clearly express efficient implementations for a suitable range of algorithms. Observability and predictability should make it possible to debug and maintain programs.

Portability: Functionality and performance, across operating systems and compilers. Parallel programming models should work on a range of targets, now and into the future.

In this book, we constrain all our examples to C and C++, and we offer the most examples in C++, since that is the language in which many new mainstream performance-oriented applications are written. We consider programming models that add parallelism support to the C and C++ languages and attempt to address the challenges of performance, productivity, and portability.

We also limit ourselves to programming models available from Intel, although, as shown in Figure 1.12, Intel actually supports a wide range of parallel programming approaches, including libraries and standards such as OpenCL, OpenMP, and MPI. The two primary shared-memory parallel programming models available from Intel are also the primary models used in this book:

Intel Threading Building Blocks (TBB): A widely used template library for C++ programmers to address most C++ needs for parallelism. TBB supports an efficient task model. TBB is available as a free, community-supported, open source version, as well as a functionally identical version with commercial support available from Intel.

Intel Cilk Plus (Cilk Plus): Compiler extensions for C and C++ to support parallelism. Cilk Plus has an efficient task model and also supports the explicit specification of vector parallelism through a set of array notations and elemental functions. Cilk Plus has both open source and commercially supported product options.

In the following, we will first discuss some desirable properties of parallel programming models, then introduce the programming models used in this book.

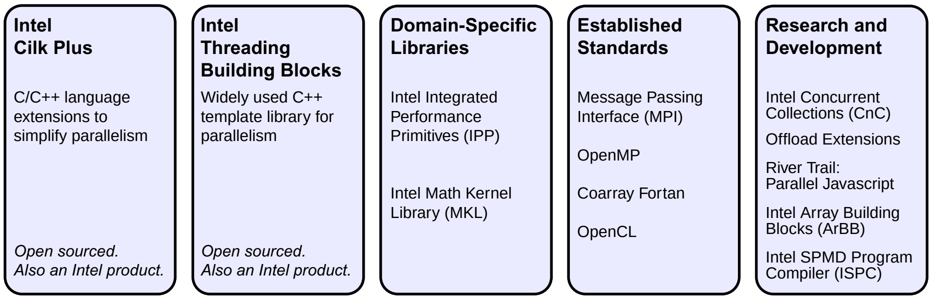
Intel Cilk Plus
C/C++ language extensions to simplify parallelism
Open sourced. Also an Intel product.
Intel Threading Building Blocks
Widely used C++ template library for parallelism
Open sourced. Also an Intel product.
Domain-Specific Libraries
Intel Integrated Performance Primitives (IPP)
Intel Math Kernel Library (MKL)
Established Standards
Message Passing Interface (MPI)
OpenMP
Coarray Fortan
OpenCL
Research and Development
Intel Concurrent Collections (CnC)
Offload Extensions
River Trail: Parallel Javascript
Intel Array Building Blocks (ArBB)
Intel SPMD Program Compiler (ISPC)

FIGURE 1.12

Parallel programming models supported by Intel. A choice of approaches is available, including pre-optimized parallel libraries; standards such as MPI, Coarray Fortran, OpenMP, and OpenCL; dynamic data-parallel virtual machines such as ArBB; domain-specific languages targeting SPMD vector parallelism such as ISPC; coordination languages such as CnC; and the primary programming models used in this book: Cilk Plus and TBB.

1.5.2 Abstractions Instead of Mechanisms

To achieve portable parallel programming you should avoid directly using hardware mechanisms. Instead, you should use abstractions that map onto those mechanisms. In particular, you should avoid **vector intrinsics** that map directly onto vector instructions and instead use array operations. You should also avoid using threads directly and program in terms of a **task** abstraction. Tasks identify only opportunities for parallelism, not the actual parallel execution mechanism. Programming should focus on the decomposition of the problem and the design of the algorithm rather than the specific mechanisms by which it will be parallelized.

There are three big reasons to avoid programming directly to specific parallel hardware mechanisms:

1. Portability is impaired severely when programming "close to the hardware."
2. Nested parallelism is important and nearly impossible to manage well using the mandatory parallelism implied by specific mechanisms such as threads.
3. Other mechanisms for parallelism, such as vectorization, exist and need to be considered. In fact, some implementations of a parallel algorithm might use threads on one machine and vectors on another, or some combination of different mechanisms.

Using abstractions for specifying vectorization rather than vector intrinsics avoids dependencies on the peculiarities of a particular vector instruction set, such as the number of elements in a vector. Even within Intel's processor product line, there are now different vector instruction set extensions with 4, 8, and 16 single-precision floating point elements per SIMD register. Fortunately there are good abstractions available to deal with these differences. For example, in both Cilk Plus and ArBB it is also possible to use either **array operations** or **elemental functions** to specify vector parallelism in a machine-independent way. OpenCL primarily depends on elemental functions. In these three cases, easily vectorized code is specified using portable abstractions.

The reasons for avoiding direct threading are more subtle, but basically a task model has less overhead, supports better composability, and gives the system more freedom to allocate resources. In particular, tasks support the specification of optional parallelism. Optional (as opposed to mandatory) parallelism supports nesting and efficient distributed **load balancing**, and can better manage converting potential to actual parallelism as needed. Nested parallelism is important for developing parallel libraries that can be used inside other parallel programs without exposing the internals of the implementation of those libraries. Such composability is fundamental to software engineering. If you want to understand more about the reasons for this shift to programming using tasks, an excellent detailed explanation of the perils of direct threading is "The Problem with Threads" [Lee06].

Tasks were the basis of an MIT research project that resulted in Cilk, the basis of Cilk Plus. This research led to the efficient work-stealing schedulers and tasking models that are now considered the best available solutions to scalable and low-overhead load balancing. TBB likewise offers an extensive set of algorithms for managing tasks using efficient, scalable mechanisms.

Cilk Plus and TBB each offer both parallel loops and parallel function invocation. The data parallel focus of ArBB generates task parallelism by allowing programmers to specify many independent operations to be run in parallel. However, ArBB does not explicitly manage tasks, leaving that to the mechanisms supplied by Cilk Plus and TBB. This also means that ArBB is composable with these models.

OpenCL is a standard based on a elemental function abstraction, and implementations vary. However, the most important pattern used by OpenCL the map pattern (the replicated execution of a single function), and we will discuss how this can be implemented efficiently.

OpenMP has several features that make it difficult to implement a built-in load balancer. It is based on loop parallelism, but unfortunately it directly exposes certain underlying aspects of its implementation. We will present some OpenMP examples in order to demonstrate that the patterns also apply to the OpenMP standard, but we recommend that new software use one of Cilk Plus or TBB to benefit from their superior composability and other advantages.

1.5.3 Expression of Regular Data Parallelism

Data parallelism is the key to achieving scalability. Merely dividing up the source code into tasks using functional decomposition will not give more than a constant factor speedup. To continue to scale to ever larger numbers of cores, it is crucial to generate more parallelism as the problem grows larger. Data parallelism achieves this, and all programming models used for examples in this book support data parallelism.

Data parallelism is a general term that actually applies to any form of parallelism in which the amount of work grows with the size of the problem. Almost all of the patterns discussed in this book, as well as the task models supported by TBB and Cilk Plus, can be used for data parallelism. However, there is a subcategory of data parallelism, **regular data parallelism**, which can be mapped efficiently onto vector instructions in the hardware, as well as to hardware threads. Use of vector instruction mechanisms can give a significant additional boost to performance. However, since vector instructions differ from processor to processor, portability requires abstractions to express such forms of data parallelism.

Abstractions built into Cilk Plus, ArBB, and OpenCL make it natural to express regular data parallelism explicitly without having to rely on the compiler inferring it. By expressing regular data parallelism explicitly, the ability of the programming model to exploit the inherent parallelism in an algorithm is enhanced.

As previously discussed, reducing everything to a serially executed procedure is a learned skill. However, such serial processing can in fact be quite unnatural for regular data-parallel problems. You are probably so used to serial programming constructs such as loops that you may not notice anymore how unnatural they can be, but the big problem for parallel programming systems is that a serial ordering of operations is in fact unnecessary in many cases. By forcing ordering of operations in a serial fashion, existing serial languages are actually removing opportunities for parallelism unnecessarily.

Consider again the simple loop shown in Listing 1.8 to add two vectors. The writer of the code probably really just meant to say "add all of the corresponding elements in `b` and `c` and put the result in

```
for (i = 0; i < 10000; ++i) {
   a[i] = b[i] + c[i];
}
```

LISTING 1.8

Serial vector addition coded as a loop in C.

```
a[0:10000] = b[0:10000] + c[0:10000];
```

LISTING 1.9

Parallel vector addition using Cilk Plus.

```
a = b + c;
```

LISTING 1.10

Parallel vector addition using ArBB.

the corresponding element of `a`." But this code implies more: It implies that the additions are done in a certain *order* as well. It might be possible for the compiler to infer that these operations can be done in parallel and do so, but it is not clear from the literal semantics of the code given that this is what is meant. Also, languages such as C and C++ make it possible to use pointers for these arrays, so in theory the data storage for `a`, `b`, and `c` could overlap or be misaligned, making it hard for the compiler to automatically use the underlying vector mechanisms effectively. For example, see Listing 1.2, which shows that unfortunately, the order does matter if the memory for the arrays in the above code could overlap.

Cilk Plus has the ability to specify data-parallel operations explicitly with new array notation extensions for C and C++. The array notations make it clear to the compiler that regular data parallelism is being specified and avoids, by specification, the above difficulties. Using array notation, we can rewrite the above loop as shown in Listing 1.9.

ArBB is even simpler, as long as the data is already stored in ArBB containers: If `a`, `b`, and `c` are all ArBB containers, the vector addition simplifies to the code shown in Listing 1.10. ArBB containers have the additional advantage that the actual data may be stored in a remote location, such as the local memory of a co-processor.

You can use these notations when you just want to operate on the elements of two arrays, and you do not care in what order the individual operations are done. This is exactly what the parallel constructs of Cilk Plus and ArBB add to C and C++. Explicit array operations such as this are not only shorter but they also get rid of the unnecessary assumption of serial ordering of operations, allowing for a more efficient implementation.

Cilk Plus, ArBB, and OpenCL also allow the specification of regular data parallelism through **elemental functions**. Elemental functions can be called in regular data parallel contexts—for example, by being applied to all the elements of an array at once. Elemental functions allow for **vectorization** by replication of the computation specified across vector lanes. In Cilk Plus, the internals of these functions are given using normal C/C++ syntax, but marked with a **pragma** and called from inside a vectorized context, such as a vectorized loop or an array slice. In ArBB, elemental functions are defined over ArBB types and called from a `map` operation—but the concept is the same. In OpenCL, elemental functions are specified in a separate C-like language. These "kernels" are then bound to data and invoked using an **application programming interface (API)**. Elemental functions are consistent with leaving the semantics of existing serial code largely intact while adding the ability to take advantage of vector mechanisms in the hardware. Both array expressions and elemental functions can also simultaneously map computations over hardware thread parallelism mechanisms.

Consider the code in Listing 1.11. Suppose the function `my_simple_add` is compiled separately, or perhaps accessed by a function pointer or virtual function call. Perhaps this function is passed in by a user to a library, and it is the library that is doing the parallel execution. Normally it would be hard for this case to be vectorized. However, by declaring `my_simple_add` as an elemental function, then it is possible to vectorize it in many of these cases. Using ArBB, it is even possible to vectorize this code in the case of function pointers or virtual function calls, since ArBB can dynamically inline code.

Getting at the parallelism in existing applications has traditionally required non-trivial rewriting, sometimes referred to as **refactoring**. Compiler technology can provide a better solution.

For example, with Cilk Plus, Listing 1.12 shows two small additions (the `__declspec(vector)` and the pragma) to Listing 1.11 that result in a program that can use either SSE or AVX instructions to yield significant speedups from vector parallelism. This will be the case even if `my_simple_add` is compiled separately and made available as a binary library. The compiler will create vectorized versions of elemental functions and call them whenever it detects an opportunity, which in this case is provided by the pragma to specify vectorization of the given loop. In the example shown, the number of calls to the function can be reduced by a factor of 8 for AVX or a factor of 4 for SSE. This can result in significant performance increases.

Another change that may be needed in order to support vectorization is conversion of data layouts from **array-of-structures** to **structure-of-arrays** (see Section 6.7). This transformation can be auto-

```
float my_simple_add(float x1, float x2) {
    return x1 + x2;
}
...
for (int j = 0; j < N; ++j) {
   outputx[j] = my_simple_add(inputa[j], inputb[j]);
}
```

LISTING 1.11

Scalar function for addition in C.

```
__declspec(vector)
float my_simple_add(float x1, float x2) {
    return x1 + x2;
}
...
#pragma simd
for (int j = 0; j < N; ++j) {
   outputx[j] = my_simple_add(inputa[j], inputb[j]);
}
```

LISTING 1.12

Vectorized function for addition in Cilk Plus.

mated by ArBB. So, while ArBB requires changes to the types used for scalar types, it can automate larger scale code transformations once this low-level rewriting has been done.

These two mechanisms, array expressions and elemental functions, are actually alternative ways to express one of the most basic parallel patterns: map. However, other regular data-parallel patterns, such as the **scan** pattern and the **reduce** pattern (discussed in Chapter 5) are also important and can also be expressed directly using the programming models discussed in this book. Some of these patterns are harder for compilers to infer automatically and so are even more important to be explicitly expressible.

1.5.4 Composability

Composability is the ability to use a feature without regard to other features being used elsewhere in your program. Ideally, every feature in a programming language is composable with every other.

Imagine if this was not true and use of an `if` statement meant you could not use a `for` statement anywhere else in an application. In such a case, linking in a library where any `if` statement was used would mean `for` statements would be disallowed throughout the rest of the application. Sounds ridiculous? Unfortunately, similar situations exist in some parallel programming models or combinations of programming models. Alternatively, the composition may be allowed but might lead to such poor performance that it is effectively useless.

There are two principal issues: incompatibility and inability to support hierarchical composition. Incompatibility means that using two parallel programming models simultaneously may lead to failures or possible failures. This can arise for many more-or-less subtle reasons, such as inconsistent use of thread-local memory. Such incompatibility can lead to failure even if the parallel regions do not directly invoke each other.

Even if two models are compatible, it may not be possible to use them in a nested or hierarchical fashion. A common case of this is when a library function is called from a region parallelized by one model, and the library itself is parallelized with a different model. Ideally a software developer should not need to know that the library was parallelized, let alone with what programming model. Having to know such details violates information hiding and separation of concerns, fundamental principles of software engineering, and leads to many practical problems. For example, suppose the library was originally serial but a new version of the library comes out that is parallelized. With models that are not composable, upgrading to the new version of this library, even if the binary interface is the same, might break the code with which it is combined.

A common failure mode in the case of nested parallelism is **oversubscription**, where each use of parallelism creates a new set of threads. When parallel routines that do this are composed hierarchically a very large number of threads can easily be created, causing inefficiencies and possibly exceeding the number of threads that the system can handle. Such soft failures can be harder to deal with than hard failures. The code might work when the system is quiet and not using a large number of threads, but fail under heavy load or when other applications are running.

Cilk Plus and TBB, the two primary programming models discussed in this book, are fully compatible and composable. This means they can be combined with each other in a variety of situations without causing failures or oversubscription. In particular, nested use of Cilk Plus with TBB is fine, as is nested use of TBB with itself or Cilk Plus with itself. ArBB can also be used from inside TBB or Cilk Plus since its implementation is based in turn on these models. In all these cases only a fixed number of threads will be created and will be managed efficiently.

These three programming models are also, in practice, compatible with OpenMP, but generally OpenMP routines should be used in a peer fashion, rather than in a nested fashion, in order to avoid over-subscription, since OpenMP creates threads as part of its execution model.

Because composability is ultimately so important, it is reasonable to hope that non-composable models will completely give way to composable models.

1.5.5 Portability of Functionality

Being able to run code on a wide variety of platforms, regardless of operating systems and processors, is desirable. The most widely used programming languages such as C, C++, and Java are portable.

All the programming models used in this book are portable. In some cases, this is because a single portable implementation is available; in other cases, it is because the programming model is a standard with multiple implementations.

TBB has been ported to a wide variety of platforms, is implemented using standard C++, and is available under an open source license. Cilk Plus is growing in adoption in compilers and is available on the most popular platforms. The Cilk Plus extensions are available in both the Intel compiler and are also being integrated into the GNU gcc compiler. Both TBB and Cilk Plus are available under open source licenses. ArBB, like TBB, is a portable C++ library and has been tested with a variety of C++ compilers. TBB and Cilk Plus are architecturally flexible and can work on a variety of modern shared-memory systems.

OpenCL and OpenMP are standards rather than specific portable implementations. However, OpenCL and OpenMP implementations are available for a variety of processors and compilers. OpenCL provides the ability to write parallel programs for CPUs as well as GPUs and co-processors.

1.5.6 Performance Portability

Portability of performance is a serious concern. You want to know that the code you write today will continue to perform well on new machines and on machines you may not have tested it on. Ideally, an application that is tuned to run within 80% of the peak performance of a machine should not suddenly run at 30% of the peak performance on another machine. However, performance portability is generally only possible with more abstract programming models. Abstract models are removed enough from the hardware design to allow programs to map to a wide variety of hardware without requiring code changes, while delivering reasonable performance relative to the machine's capability.

Of course, there are acceptable exceptions when hardware is considered exotic. However, in general, the more flexible and abstract models can span a wider variety of hardware.

Cilk Plus, TBB, OpenMP, and ArBB are designed to offer strong performance portability. OpenCL code tends to be fairly low level and as such is more closely tied to the hardware. Tuning OpenCL code tends to strongly favor one hardware device over another. The code is (usually) still functionally portable but may not perform well on devices for which it was not tuned.

1.5.7 Safety, Determinism, and Maintainability

Parallel computation introduces several complications to programming, and one of those complications is non-determinism. **Determinism** means that every time the program runs, the answer is the same. In serial computation, the order of operations is fixed and the result is naturally deterministic. However, parallel programs are not naturally deterministic. The order of operation of different threads may be interleaved in an arbitrary order. If those threads are modifying shared data, it is possible that different runs of a program may produce different results even with the same input. This is known, logically enough, as **non-determinism**. In practice, the randomness in non-deterministic parallel programs arises from the randomness of thread scheduling, which in turn arises from a number of factors outside the control of the application.

Non-determinism is not necessarily bad. It is possible, in some situations, for non-deterministic algorithms to outperform deterministic algorithms. However, many approaches to application testing assume determinism. For example, for non-deterministic programs testing tools cannot simply compare results to one known good solution. Instead, to test a non-deterministic application, it is necessary to prove that the result is correct, since different but correct results may be produced on different runs. This may be as simple as testing against a tolerance for numerical applications, but may be significantly more involved in other cases. Determinism or repeatability may even be an application requirement (for example, for legal reasons), in which case you will want to know how to achieve it.

Non-determinism may also be an error. Among the possible interleavings of threads acting on shared data, some may be incorrect and lead to incorrect results or corrupted data structures. The problem of safety is how to ensure that only correct orderings occur.

One interesting observation is that many of the parallel patterns used in this book are either deterministic by nature or have deterministic variants. Therefore, one way to achieve complete determinism is to use only the subset of these patterns that are deterministic. An algorithm based on a composition of deterministic patterns will be deterministic. In fact, the (unique) result of each deterministic pattern can be made equivalent to some serial ordering, so we can also say that such programs are **serially consistent**—they always produce results equivalent to some serial program. This makes debugging and reasoning about such programs much simpler.

Of the programming models used in this book, ArBB in particular emphasizes determinism. In the other models, determinism can (usually) be achieved with some discipline. Some performance may be lost by insisting on determinism, however. How much performance is lost will depend on the algorithm. Whether a non-deterministic approach is acceptable will necessarily be decided on a case-by-case basis.

1.5.8 Overview of Programming Models Used

We now summarize the basic properties of the programming models used in this book.

Cilk Plus

The Cilk Plus programming model provides the following features:

- Fork–join to support irregular parallel programming patterns and nesting
- Parallel loops to support regular parallel programming patterns, such as map
- Support for explicit vectorization via array sections, `pragma simd`, and elemental functions

- **Hyperobjects** to support efficient reduction
- Serial semantics if keywords are ignored (also known as **serial elision**)
- Efficient load balancing via work-stealing

The Cilk Plus programming model is integrated with a C/C++ compiler and extends the language with the addition of keywords and array section notation.

The Cilk (pronounced "silk") project originated in the mid-1990s at M.I.T. under the guidance of Professor Charles E. Leiserson. It has generated numerous papers, inspired a variety of "work stealing" task-based schedulers (including TBB, Cilk Plus, TPL, PPL and GCD), has been used in teaching, and is used in some university-level textbooks.

Cilk Plus evolved from Cilk and provides very simple but powerful ways to specify parallelism in both C and C++. The simplicity and power come, in no small part, from being embedded in the compiler. Being integrated into the compiler allows for a simple syntax that can be added to existing programs. This syntax includes both array sections and a small set of keywords to manage fork–join parallelism.

Cilk started with two keywords and a simple concept: the asynchronous function call. Such a call, marked with the keyword `cilk_spawn`, is like a regular function call except that the caller can keep going in parallel with the callee. The keyword `cilk_sync` causes the current function to wait for all functions that it spawned to return. Every function has an implicit `cilk_sync` when it returns, thus guaranteeing a property similar to plain calls: When a function returns, the entire call tree under it has completed.

Listings 1.13 and 1.14 show how inserting a few of these keywords into serial code can make it parallel. The classic recursive function to compute Fibonacci numbers serves as an illustration. The addition of one `cilk_spawn` and one `cilk_sync` allows parallel execution of the two recursive calls, waiting for them to complete, and then summing the results afterwards. Only the first recursive call is spawned, since the caller can do the second recursive call.

This example highlights the key design principle of Cilk: A parallel Cilk program is a serial program with keyword "annotations" indicating where parallelism is permitted (but not mandatory). Furthermore there is a strong guarantee of serial equivalence: In a well-defined Cilk program, the parallel program computes the same answer as if the keywords are ignored. In fact, the Intel implementation of Cilk Plus ensures that when the program runs on one processor, operations happen in the same order as the equivalent serial program. Better yet, the serial program can be recovered using the preprocessor; just `#define cilk_spawn` and `cilk_sync` to be whitespace. This property enables Cilk code to be compiled by compilers that do not support the keywords.

Since the original design of Cilk, one more keyword was added: `cilk_for`. Transforming a loop into a parallel loop by changing `for` to `cilk_for` is often possible and convenient. Not all serial loops can be converted this way; the iterations must be independent and the loop bounds must not be modified in the loop body. However, within these constraints, many serial loops can still be parallelized. Conversely, `cilk_for` can always be replaced with `for` by the preprocessor when necessary to obtain a serial program.

The implementation of `cilk_for` loops uses a recursive approach (Section 8.3) that spreads overhead over multiple tasks and manages **granularity** appropriately. The alternative of writing a serial `for` loop that spawns each iteration is usually much inferior, because it puts all the work of spawning on a single task and bottlenecks the load balancing mechanism, and a single iteration may be too small to justify spawning it as a separate task.

```
int fib (int n) {
   if (n < 2) {
      return n;
   } else {
      int x, y;
      x = fib(n - 1);
      y = fib(n - 2);
      return x + y;
   }
}
```

LISTING 1.13

Serial Fibonacci computation in C. It uses a terribly inefficient algorithm and is intended only for illustration of syntax and semantics.

```
int fib (int n) {
   if (n < 2) {
      return n;
   } else {
      int x, y;
      x = cilk_spawn fib(n - 1);
      y = fib(n - 2);
      cilk_sync;
      return x + y;
   }
}
```

LISTING 1.14

Parallel Cilk Plus variant of Listing 1.13.

Threading Building Blocks (TBB)

The Threading Building Blocks (TBB) programming model supports parallelism based on a tasking model. It provides the following features:

- Template library supporting both regular and irregular parallelism
- Direct support for a variety of parallel patterns, including map, fork–join, task graphs, reduction, scan, and pipelines
- Efficient work-stealing load balancing
- A collection of thread-safe data structures
- Efficient low-level primitives for atomic operations and memory allocation

TBB is a library, not a language extension, and thus can be used with with any compiler supporting ISO C++. Because of that, TBB uses C++ features to implement its "syntax." TBB requires the use of function objects (also known as **functors**) to specify blocks of code to run in parallel. These were somewhat tedious to specify in C++98. However, the C++11 addition of **lambda expressions** (see Appendix D) greatly simplifies specifying these blocks of code, so that is the style used in this book.

TBB relies on templates and generic programming. Generic programming means that algorithms are written with the fewest possible assumptions about data structures, which maximizes potential for reuse. The C++ Standard Template Library (STL) is a good example of generic programming in which the interfaces are specified only by requirements on template types and work across a broad range of types that meet those requirements. TBB follows a similar philosophy.

Like Cilk Plus, TBB is based on programming in terms of tasks, not threads. This allows it to reduce overhead and to more efficiently manage resources. As with Cilk Plus, TBB implements a common thread pool shared by all tasks and balances load via work-stealing. Use of this model allows for nested parallelism while avoiding the problem of over-subscription.

The TBB implementation generally avoids global locks in its implementation. In particular, there is no global task queue and the memory allocator is lock free. This allows for much more scalability. As discussed later, global locks effectively serialize programs that could otherwise run in parallel.

Individual components of TBB may also be used with other parallel programming models. It is common to see the TBB parallel memory allocator used with Cilk Plus or OpenMP programs, for example.

OpenMP

The OpenMP programming model provides the following features:

- Creation of teams of threads that jointly execute a block of code
- Conversion of loops with bounded extents to parallel execution by a team of threads with a simple annotation syntax
- A tasking model that supports execution by an explicit team of threads
- Support for atomic operations and locks
- Support for reductions, but only with a predefined set of operations

The OpenMP interface is based on a set of compiler directives or pragmas in Fortran, C and C++ combined with an API for thread management. In theory, if the API is replaced with a stub library and the pragmas are ignored then a serial program will result. With care, this serial program will produce a result that is the “same” as the parallel program, within numerical differences introduced by reordering of floating-point operations. Such reordering, as we will describe later, is often required for parallelization, regardless of the programming model.

OpenMP is a standard organized by an independent body called the OpenMP Architecture Review Board. OpenMP is designed to simplify parallel programming for application programmers working in high-performance computing (HPC), including the parallelization of existing serial codes. Prior to OpenMP (first released in 1997), computer vendors had distinct directive-based systems. OpenMP standardized common practice established by these directive-based systems. OpenMP is supported by most compiler vendors including the GNU compilers and other open source compilers.

The most common usage of OpenMP is to parallelize loops within a program. The pragma syntax allows the reinterpretation of loops as parallel operations, which is convenient since the code inside the loop can still use normal Fortran, C, or C++ syntax and memory access. However, it should be noted that (as with Cilk Plus) only loops that satisfy certain constraints can be annotated and converted into parallel structures. In particular, iteration variable initialization, update, and termination tests must be one of a small set of standard forms, it must be possible to compute the number of iterations in advance, and the loop iterations must not depend on each other. In other words, a "parallel loop" in OpenMP implements the **map** pattern, using the terminology of this book. In practice, the total number of iterations is broken up into blocks and distributed to a team of threads.

OpenMP implementations do not, in general, check that loop iterations are independent or that race conditions do not exist. As with Cilk Plus, TBB, and OpenCL, avoiding incorrect parallelizations is the responsibility of the programmer.

The main problem with OpenMP for mainstream users is that OpenMP exposes the threads used in a computation. Teams of threads are explicit and must be understood to understand the detailed meaning of a program. This constrains the optimizations available from the OpenMP runtime system and makes the tasking model within OpenMP both more complex to understand and more challenging to implement.

The fact that threads are exposed encourages a programmer to think of the parallel computation in terms of threads and how they map onto cores. This can be an advantage for algorithms explicitly designed around a particular hardware platform's memory hierarchy, which is common in HPC. However, in more mainstream applications, where a single application is used on a wide range of hardware platforms, this can be counterproductive. Furthermore, by expressing the programming model in terms of explicit threads, OpenMP encourages (but does not require) algorithm strategies based on explicit control over the number of threads. On a dedicated HPC machine, having the computation depend upon or control the number of threads may be desirable, but in a mainstream application it is better to let the system decide how many threads are appropriate.

The most serious problem caused by the explicit threading model behind OpenMP is the fact that it limits the ability of OpenMP to compose with itself. In particular, if an OpenMP parallel region creates a team of threads and inside that region a library is called that also uses OpenMP to create a team of threads, it is possible that n^2 threads will be created. If repeated (for example, if recursion is used) this can result in exponential oversubscription. The resulting explosion in the number of threads created can easily exhaust the resources of the operating system and cause the program to fail. However, this only happens if a particular OpenMP option is set: `OMP_NESTED=TRUE`. Fortunately the default is `OMP_NESTED=FALSE`, and it should generally be left that way for mainstream applications. When OpenMP and a model like TBB or Cilk Plus are nested and the default setting `OMP_NESTED=FALSE` is used, at worst $2p$ workers will be created, where p is the number of cores. This can be easily managed by the operating system.

It is also recommended to use `OMP_WAIT_POLICY=ACTIVE` and `OMP_DYNAMIC=TRUE` to enable dynamic scheduling. Using static scheduling in OpenMP (`OMP_DYNAMIC=FALSE`) is not recommended in a mainstream computing environment, since it assumes that a fixed number of threads will be used from one parallel region to the next. This constrains optimizations the runtime system may carry out.

HPC programmers often use OpenMP to explicitly manage a team of threads using the thread ID available through the OpenMP API and the number of threads to control how work is mapped to threads. This also limits what the runtime system can do to optimize execution of the threads. In particular, it limits the ability of the system to perform load balancing by moving work between threads. TBB and Cilk Plus intentionally do not include these features.

In OpenMP, point-to-point synchronization is provided through low-level (and error-prone) locks. Another common synchronization construct in OpenMP is the **barrier**. A classical barrier synchronizes a large number of threads at once by having all threads wait on a lock until all other threads arrive at the same point. In Cilk Plus and TBB, where similar constructs exist (for example, implicitly at the end of a `cilk_for`), they are implemented as pairwise joins, which are more scalable.

Array Building Blocks (ArBB)

The Array Building Blocks (ArBB) programming model supports parallelization by the specification of sequences of data-parallel operations. It provides the following features:

- High-level data parallel programming with both elemental functions and vector operations
- Efficient **collective operations**
- Automatic fusion of multiple operations into more intensive kernels
- Dynamic code generation under programmer control
- Offload to **attached co-processors** without change to source code
- Deterministic by default, safe by design

ArBB is compiler independent and, like TBB, in conjunction with its embedded C++ front-end can in theory be used with any ISO C++ compiler. The vectorized code generation supported by its virtual machine library is independent of the compiler it is used with.

Array Building Blocks is the most high level of the models used in this book. It does not explicitly depend on tasks in its interface, although it does use them in its implementation. Instead of tasks, parallel computations are expressed using a set of operations that can act over collections of data. Computations can be expressed by using a sequence of parallel operations, by replicating elemental **functions** over the elements of a collection, or by using a combination of both.

Listing 1.15 shows how a computation in ArBB can be expressed using a sequence of parallel operations, while Listing 1.16 shows how the same operation can be expressed by replicating a function over a collection using the `map` operation. In addition to per-element vector operations, ArBB also supports a set of collective and data-reorganization operations, many of which map directly onto patterns discussed in later chapters.

```
void arbb_vector (
    dense<f32>& A,
    dense<f32> B,
    dense<f32> C,
    dense<f32> D
) {
    A += B - C/D;
}

dense<f32> A, B, C, D;
// fill A, B, C, D with data ...

// invoke function over entire collections
call(arbb_vector)(A,B,C,D);
```

LISTING 1.15

Vector computation in ArBB.

```
void arbb_map (
   f32& a, // input and output
   f32 b,  // input
   f32 c,  // input
   f32 d   // input
) {
   a += b − c/d;
}

void arbb_call (
   dense<f32>& A,  // input and output
   dense<f32> B,   // input
   dense<f32> C,   // input
   f32 d           // input (uniform; will be replicated )
) {
   map(arbb_map)(A,B,C,d);
}
```

LISTING 1.16

Elemental function computation in ArBB.

ArBB manages data as well as code. This has two benefits: Data can be laid out in memory for better vectorization and data locality, and data and computation can be offloaded to attached co-processors with no changes to the code. It has the disadvantage that extra code is required to move data in and out of the data space managed by ArBB, and extra data movement may be required.

OpenCL

The OpenCL programming model provides the following features:

- Ability to offload computation and data to an attached co-processor with a separate memory space
- Invocation of a regular grid of parallel operations using a common kernel function
- Support of a task queue for managing asynchronous kernel invocations

The OpenCL programming model includes both a kernel language for specifying kernels and an API for managing data transfer and execution of kernels from the host. The kernel language is both a superset and a subset of C99, in that it omits certain features, such as `goto`, but includes certain other features, such as a "swizzle" notation for reordering the elements of short vectors.

OpenCL is a standard organized by Khronos and supported by implementations from multiple vendors. It was primarily designed to allow offload of computation to GPU-like devices, and its memory and task grouping model in particular reflects this. In particular, there are explicit mechanisms for allocating local on-chip memory and for sharing that memory between threads in a workgroup. However, this sharing and grouping are not arranged in an arbitrary hierarchy, but are only one level deep, reflecting the hardware architecture of GPUs. However, OpenCL can also in theory be used for other co-processors as well as CPUs.

The kernel functions in OpenCL corresponds closely to what we call "elemental functions," and kernel invocation corresponds to the map pattern described in this book.

OpenCL is a relatively low-level interface and is meant for performance programming, where the developer must specify computations in detail. OpenCL may also be used by higher level tools as a target language. The patterns discussed in this book can be used with OpenCL but few of these patterns are reflected directly in OpenCL features. Instead, the patterns must be reflected in algorithm structure and conventions.

As a low-level language, OpenCL provides direct control over the host and the compute devices attached to the host. This is required to support the extreme range of devices addressed by OpenCL: from CPUs and GPUs to embedded processors and field-programmable gate arrays (FPGAs). However, OpenCL places the burden for performance portability on the programmer's shoulders. Performance portability is possible in OpenCL, but it requires considerable work by the programmer, often to the point of writing a different version of a single kernel for each class of device.

Also, OpenCL supports only a simple two-level memory model, and for this and other reasons (for example, lack of support for nested parallelism) it lacks composability.

In placing OpenCL in context with the other programming models we have discussed, it is important to appreciate the goals for the language. OpenCL was created to provide a low-level "hardware abstraction layer" to support programmers needing full control over a heterogeneous platform. The low-level nature of OpenCL was a strategic decision made by the group developing OpenCL. To best support the emergence of high-level programming models for heterogeneous platforms, first a portable hardware abstraction layer was needed.

OpenCL is not intended for mainstream programmers the way TBB, Cilk Plus, or OpenMP are. Lacking high-level programming models for heterogeneous platforms, application programmers often turn to OpenCL. However, over time, higher level models will likely emerge to support mainstream application programmers and OpenCL will be restricted to specialists writing the runtimes for these higher level models or for detailed performance-oriented libraries.

However, we have included it in this book since it provides an interesting point of comparison.

1.5.9 When to Use Which Model?

When multiple programming models are available, the question arises: When should which model be used? As we will see, TBB and Cilk Plus overlap significantly in functionality, but do differ in deployment model, support for vectorization, and other factors. OpenCL, OpenMP, and ArBB are each appropriate in certain situations.

Cilk Plus can be used whenever a compiler supporting the Cilk Plus extensions, such as the Intel C++ compiler or gcc, can be used. It targets both hardware thread and vector mechanisms in the processor and is a good all-around solution. It currently supports both C and C++.

Threading Building Blocks (TBB) can be used whenever a compiler-portable solution is needed. However, TBB does not, itself, do vectorization. Generation of vectorized code must be done by the compiler TBB is used with. TBB does, however, support **tiling** ("blocking") and other constructs so that opportunities for vectorization are exposed to the underlying compiler.

TBB and Cilk Plus are good all-around models for C++. They differ mainly in whether a compiler with the Cilk Plus extensions can be used. We also discuss several other models in this book, each of which may be more appropriate in certain specific circumstances.

OpenMP is nearly universally available in Fortran, C, and C++ compilers. It has proven both popular and effective with scientific code, where any shortcomings in composability tend to be unimportant because of the dominance of intense computational loops as opposed to complex nested parallelism. Also, the numerous options offered for OpenMP are highly regarded for the detailed control they afford for the difficult task of tuning supercomputer code.

Array Building Blocks can be used whenever a high-level solution based on operations on collections of data is desired. It supports dynamic code generation, so it is compiler independent like TBB but supports generation of vectorized code like Cilk Plus.

Because of its code generation capabilities, ArBB can also be used for the implementation of custom parallel languages, a topic not discussed at length in this book. If you are interested in this use of ArBB, please see the online documentation for the ArBB Virtual Machine, which provides a more suitable interface for this particular application of ArBB than the high-level C++ interface used in this book. ArBB can also be used to offload computation to co-processors.

OpenCL provides a standard solution for offloading computation to GPUs, CPUs, and accelerators. It is rather low level and does not directly support many of the patterns discussed in this book, but many of them can still be implemented. OpenCL tends to use minimal abstraction on top of the physical mechanisms.

OpenMP is also standard and is available in many compilers. It can be used when a solution is needed that spans compilers from multiple vendors. However, OpenMP is not as composable as Cilk Plus or TBB. If nested parallelism is needed, Cilk Plus or TBB would be a better choice.

1.6 ORGANIZATION OF THIS BOOK

This chapter has provided an introduction to some key concepts and described the motivation for studying this book. It has also provided a basic introduction to the programming models that we will use for examples.

Chapter 2 includes some additional background material on computer architecture and performance analysis and introduces the terminology and conventions to be used throughout this book.

Chapters 3 to 9 address the most important and common parallel patterns. Gaining an intuitive understanding of these is fundamental to effective parallel programming. Chapter 3 provides a general overview of all the patterns and discusses serial patterns and the relationship of patterns to structured programming. Chapter 4 explains map, the simplest and most scalable parallel pattern and one of the first that should be considered. Chapter 5 discusses collective patterns such as reduce and scan. Collectives address the need to combine results from map operations while maintaining the benefits of parallelism. Chapter 6 discusses data reorganization. Effective data management is often the key to efficient parallel algorithms. This chapter also discusses some memory-related optimizations, such as conversion of array-of-structures to structures-of-arrays. Chapter 8 explains the fork–join pattern and its relationship to tasks. This pattern provides a method to subdivide a problem recursively while distributing overhead in an efficient fashion. This chapter includes many detailed examples, including discussions of how to implement other patterns in terms of fork–join. Chapter 9 discusses the pipeline pattern, where availability of data drives execution rather than control flow.

The remainder of the chapters in the book consist of examples to illustrate and extend the fundamentals from earlier chapters.

The appendices include a list of further reading and self-contained introductions to the primary programming models used in this book.

1.7 SUMMARY

In this chapter, we have described recent trends in computer architecture that are driving a need for explicit parallel programming. These trends include a continuation of Moore's Law, which is leading to an exponentially growing number of transistors on integrated devices. Three other factors are limiting the potential for non-parallelized applications to take advantage of these transistors: the power wall, the ILP (instruction-level-parallelism) wall, and the memory wall. The power wall means that clock rates cannot continue to scale without exceeding the air-cooling limit. The ILP wall means that, in fact, we are *already* taking advantage of most low-level parallelism in scalar code and do not expect any major improvements in this area. We conclude that explicit parallel programming is likely necessary due to the significant changes in approach needed to achieve scalability. Finally, the memory wall limits performance since the bandwidth and latency of communication are improving more slowly than the capability to do computation. The memory wall affects scalar performance but is also a major factor in the scalability of parallel computation, since communication between processors can introduce overhead and latency. Because of this, it is useful to consider the memory and communication structure of an algorithm even before the computational structure.

In this book, we take a structured approach to parallel computation. Specifically, we describe a set of patterns from which parallel applications can be composed. Patterns provide a vocabulary and a set of best practices for describing parallel applications. The patterns embody design principles that will help you design efficient and scalable applications.

Throughout this book, we give many examples of parallel applications. We have chosen to use multiple parallel programming models for these examples, but with an emphasis on TBB and Cilk Plus. These models are portable and also provide high performance and portability. However, by using multiple programming models, we seek to demonstrate that the patterns we describe can be used in a variety of programming systems.

When designing an algorithm, it is useful as you consider various approaches to have some idea of how each possible approach would perform. In the next chapter, we provide additional background especially relevant for predicting performance and scalability. First, we describe modern computer architectures at a level of detail sufficient for this book, with a focus on the key concepts needed for predicting performance. Then, we describe some classic performance models, including Amdahl's Law and Gustafson-Barsis' Law. These laws are quite limited in predictive power, so we introduce another model, the work-span model, that is much more accurate at predicting scalability.

APPENDIX

TBB

C

This appendix provides a concise introduction to Intel Threading Building Blocks (Intel TBB). It covers the subset used by this book. A good introduction is available in the O'Reilly Nutshell Book on TBB, which covers the essentials of TBB [Rei07]. The book was published in 2007 when TBB version 2.0 appeared, so some newer features are not covered. It is nevertheless a solid introduction to TBB. For a more complete guide, see the TBB Reference, Tutorial, and Design Patterns documents, which can be downloaded from `http://threadingbuildingblocks.org/`.

TBB is a collection of components that outfits C++ for parallel programming. Figure C.1 illustrates these components. At the heart of TBB is a task scheduler, which is most often used indirectly via the parallel algorithms in TBB, such as `tbb::parallel_for`. The rest of TBB provides thread-aware memory allocation, portable synchronization primitives, scalable containers, and a variety of useful utilities. Each part is important for parallelism. Indeed the non-tasking features are intended for use with other parallelism frameworks such as Cilk Plus, ArBB, and OpenMP, so that those frameworks do not have to duplicate key functionality.

C.1 UNIQUE CHARACTERISTICS

TBB shares many of the key attributes of Cilk Plus as enumerated in Section B.1, but it differs form Cilk Plus on several points:

- TBB is designed to work without any compiler changes, and thus be quickly portable to new platforms. As a result, TBB has been ported to a multitude of key operating systems and processors, and code written with TBB can likewise be easily ported.
- As a consequence of avoiding any need for compiler support, TBB does not have direct support for vector parallelism. However, TBB combined with array notation or `#pragma simd` from Cilk Plus or auto-vectorization can be an effective tool for exploiting both thread and vector parallelism.
- TBB is designed to provide comprehensive support for C++ developers in one package. It supports multiple paradigms of parallel programming. It goes beyond the strict fork–join model of Cilk Plus by supporting pipelines, dataflow, and unstructured task graphs. The additional power that these features bring is sometimes worth the additional complexity they bring to a program.
- TBB is intended to provide low-level services such as memory allocation and atomic operations that can be used by programs using other frameworks, including Cilk Plus.

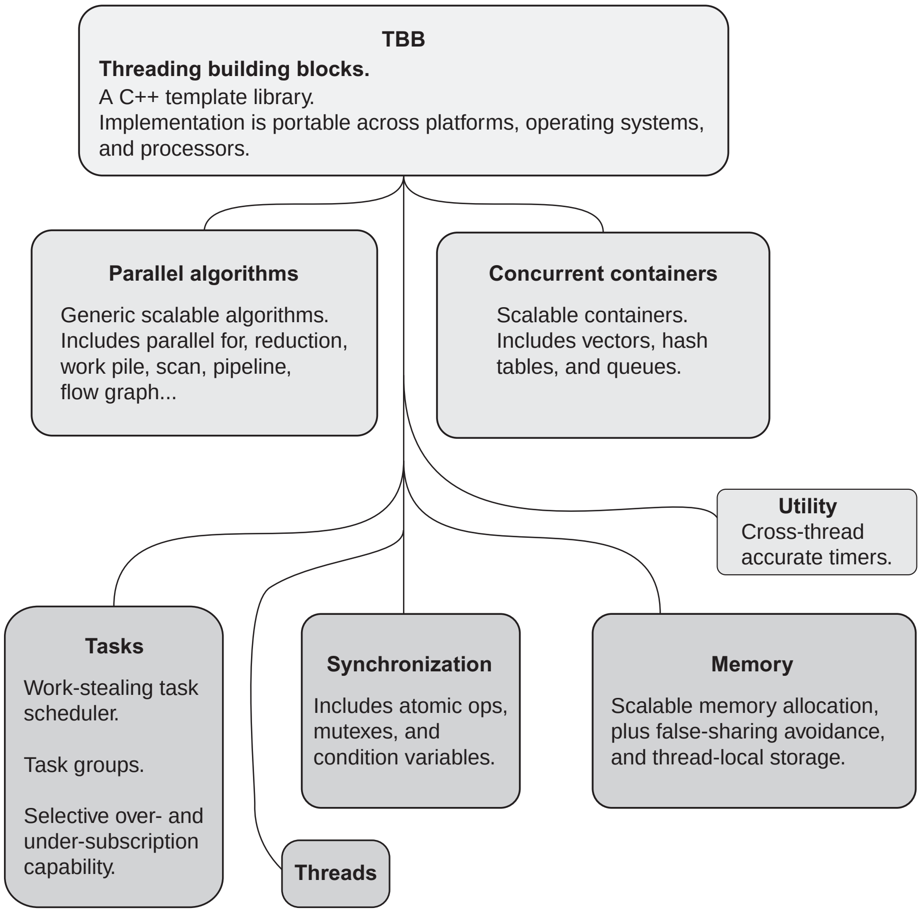
TBB
Threading building blocks.
A C++ template library.
Implementation is portable across platforms, operating systems, and processors.
Parallel algorithms
Generic scalable algorithms. Includes parallel for, reduction, work pile, scan, pipeline, flow graph...
Concurrent containers
Scalable containers. Includes vectors, hash tables, and queues.
Utility
Cross-thread accurate timers.
Tasks
Work-stealing task scheduler.
Task groups.
Selective over- and under-subscription capability.
Synchronization
Includes atomic ops, mutexes, and condition variables.
Memory
Scalable memory allocation, plus false-sharing avoidance, and thread-local storage.
Threads

FIGURE C.1

Overview of Threading Building Blocks.

TBB is an active open source project. It is widely adopted and often cited in articles about parallelism in C++. It continues to grow as the parallel ecosystem evolves.

C.2 USING TBB

Include the header `<tbb/tbb.h>` to use TBB in a source file. All public identifiers are in namespace `tbb` or `tbb::flow`. In the following descriptions, the phrase “in parallel” indicates that parallelism is permitted if resources allow, but is not mandated. As with Cilk Plus, the license to ignore unnecessary parallelism allows the TBB task scheduler to use parallelism efficiently.

C.3 parallel_for

The function template `parallel_for` maps a functor across range of values. The template takes several forms. The simplest is:

```
tbb::parallel_for(first,last,f)
```

where f is a functor. It evaluates the expression $f(i)$ in parallel for all i in the half-open interval [*first*,*last*), Both *first* and *last* must be of the same integral type. It is a parallel equivalent of:

```
for (auto i=first; i<last; ++i) f(i);
```

A slight variation specifies a stride:

```
tbb::parallel_for(first,last,stride,f)
```

It is like the previous version, except that the possible values of i step by *stride*, starting with *first*. This form is a parallel equivalent of:

```
for (auto i=first; i<last; i+=stride ) f(i);
```

Another form of `parallel_for` takes two arguments:

```
tbb::parallel_for(range,f)
```

It decomposes *range* into subranges and applies f to each subrange, in parallel. Hence, the programmer has the opportunity to optimize f to operate on an entire subrange instead of a single index. This version in effect exposes the tiled implementation of the map pattern used by TBB.

This form of parallel for also generalizes the parallel map pattern beyond one-dimensional ranges. The argument *range* can be any *recursively splittable range* type. A type `R` is such a type if it has the following methods:

`R::R(const R&)`	Copy constructor.
`R:: R()`	Destructor.
`bool R::is_divisible() const`	True if splitting constructor can be called, false otherwise.
`bool R::empty() const`	True if range is empty, false otherwise.
`R::R(R& r, split)`	Splitting constructor. It splits range `r` into two subranges. One of the subranges is the newly constructed range. The other subrange is overwritten onto `r`.

The implementation of `parallel_for` uses these methods to implement a generic recursive map in the spirit of Listing 8.1.

C.3.1 blocked_range

The most commonly used recursive range is `tbb::blocked_range`. It is typically used with integral types or random-access iterator types. For example, `blocked_range<int>(0,8)` represents the index range $\{0,1,2,3,4,5,6,7\}$. An optional third argument called the *grainsize* specifies the maximum size for splitting. It defaults to 1. For example, the following snippet splits a range of size 30 with grainsize 20 into two indivisible subranges of size 15:

```
// Construct half-open interval [0,30) with grainsize of 20
blocked_range<int> r(0,30,20);
assert(r.is_divisible());
// Call splitting constructor
blocked_range<int> s(r);
// Now r=[0,15) and s=[15,30) and both have a grainsize 20
// Inherited from the original value of r
assert(!r.is_divisible());
assert(!s.is_divisible());
```

Listing 4.2 on page 126 shows an example that uses `blocked_range` with `parallel_for`.

A two-dimensional variant is called `tbb::blocked_range2d`. It permits using a single `parallel_for` to iterate over two dimensions at once, which sometimes yields better cache behavior than nesting two one-dimensional instances of `parallel_for`.

C.3.2 Partitioners

The range form of `parallel_for` takes an optional *partitioner* argument, which lets the programmer specify performance-related tactics [RVK08]. The argument can have one of three types:

- `auto_partitioner`: The runtime will try to subdivide the range sufficiently to balance load, but no further. This behavior is the same as when no partitioner is specified.
- `simple_partitioner`: The runtime must subdivide the range into subranges as finely as possible; that is, method `is_divisible` will be false for the final subranges.
- `affinity_partitioner`: Request that the assignment of subranges to underlying threads be similar to a previous invocation of `parallel_for` or `parallel_reduce` with the same `affinity_partitioner` object.

These partitioners also work with `parallel_reduce`.

An invocation of `parallel_for` with a `simple_partitioner` looks like:

```
parallel_for(r,f,simple_partitioner());
```

This partitioner is useful in two scenarios:

- The functor f uses a fixed amount of memory for temporary storage, and hence cannot deal with subranges of arbitrary size. For example, if r is a `blocked_range`, the partitioner guarantees that f is invoked on subranges not exceeding the grainsize of r.
- The work for $f(r)$ is highly unbalanced in a way that fools the `auto_partitioner` heuristic into not dividing work finely enough to balance load.

An `affinity_partitioner` can be used for **cache fusion** (Section 4.4). Unlike the other two partitioners, it carries state. The state holds information for replaying the assignment of subranges to threads. Listing C.1 shows an example of its use in a common pattern: serially iterating a map. In the listing, variable `ap` enables cache fusion of each map to the next map. Because it is carrying information between serial iterations, it must be declared outside the serial loop.

```
void relax(
   double* a, // Pointer to array of data
   double* b, // Pointer to temporary storage
   size_t n,   // Number of data elements
   int iterations //Number of serial iterations
) {
   assert(iterations%2==0);
   // Partitioner should be declared outside the loop
   tbb::affinity_partitioner ap;
   // Serial loop around a parallel loop
   for( size_t t=0; t<iterations; ++t ) {
      tbb::parallel_for(
         tbb::blocked_range<size_t>(1,n-1),
         [=]( tbb::blocked_range<size_t> r ) {
            size_t e = r.end();
#pragma simd
            for( size_t i=r.begin(); i<e; ++i )
               b[i] = (a[i-1]+a[i]+a[i+1])*(1/3.0);
         },
         ap);
      std::swap(a,b);
   }
}
```

LISTING C.1

Example of `affinity_partitioner`. TBB uses the variable `ap` to remember on which threads ran which subranges of the previous invocation of `parallel_for` and biases execution toward replaying that assignment. The `pragma simd` is for showmanship. It makes the impact of the partitioner more dramatic by raising arithmetic performance so that memory bandwidth becomes the limiting resource.

C.4 parallel_reduce

Function template `parallel_reduce` performs a reduction over a recursive range. It has several forms. The form used in this book is:

```
T result = tbb::parallel_reduce(
   range,
   identity,
   subrange_reduction,
   combine);
```

where:

- `range` is a recursive range as for `parallel_for`, such as `blocked_range`.
- `identity` is the identity element of type *T*. The type of this argument determines the type used to accumulate the reduction value, so be careful about what type it has.

- `subrange_reduction` is a functor such that *subrange_reduction*(*subrange*,*init*) returns a reduction value over *init* and *subrange*. The type of *subrange* is the type of the *range* argument to `parallel_reduce`. The type of *init* is T, and the returned reduction value must be convertible to type T. Do not forget to include the contribution of *init* to the reduction value.
- `combine` is a functor such that *combine*(x,y) takes two arguments of type T and returns a reduction value for them. This function must be associative but does not need to be commutative.

Listings 5.5 and 5.6 in Section 5.3.4 show example invocations. The latter listing demonstrates how to do accumulation at higher precision than the values being reduced.

An alternative way to do reduction is via class `tbb::enumerable_thread_specific`, as demonstrated in Section 11.3. General advice on which to use:

- If type T takes little space and is cheap to copy, or the combiner operation is non-commutative, use `parallel_reduce`.
- If type T is large and expensive to copy *and* the combiner operation is commutative, use `enumerable_thread_specific`.

C.5 parallel_deterministic_reduce

Template function `parallel_deterministic_reduce` is a variant of `parallel_reduce` that is **deterministic** even when the combiner operation is non-associative. The result is *not* necessarily the same as left-to-right serial reduction, even when executed with a single worker, because the template uses a fixed tree-like reduction order for a given input.

As of this writing, `parallel_deterministic_reduce` is a "preview feature" that must be enabled by setting the preprocessory symbol `TBB_PREVIEW_DETERMINISTIC_REDUCE=1` either on the compiler command line or *before* including TBB headers in a source file.

C.6 parallel_pipeline

Template function `parallel_pipeline` is used for building a **pipeline** of serial and parallel stages. See Section 9.4.1 for details and Listing 12.2 for an example.

C.7 parallel_invoke

Template function `parallel_invoke` evaluates a fixed set of functors in parallel. For example,

```
tbb::parallel_invoke(f,g,h);
```

evaluates the expressions `f()`, `g()`, and `h()` in parallel and waits until they all complete. Anywhere from 2 to 10 functors are currently supported. Listing 13.3 (page 302) and Listing 15.4 (page 322) show uses of `parallel_invoke`. Both listings cross-reference similar code in Cilk Plus, so you can compare the syntactic difference.

C.8 task_group

Class task_group runs an arbitrary number of functors in parallel. Listing C.2 shows an example.

In general, a single task_group should not be used to run a large number of tasks, because it can become a sequential bottleneck. Consider using parallel_for for a large number of tasks.

If one of the functors throws an exception, the task group is cancelled. This means that any tasks in the group that have not yet started will not start at all, but all currently running tasks will keep going. After all running tasks in the group complete, one of the exceptions thrown by the tasks will be thrown to the caller of wait. Hence, if nested parallelism is created by nesting task_group, the exception propagates up the task tree until it is caught.

Listing 8.12 on page 235 shows a use of task_group.

C.9 task

Class tbb::task is the lowest-level representation of a task in TBB. It is designed primarily for efficient execution, not convenience, because it serves as a foundation, and thus should impose minimal performance penalty. Higher level templates such as parallel_for and task_group provide

```
// Item in a linked list
class morsel {
public:
   void munch();
   morsel* next;
};

// Apply method munch to each item in a linked
// list rooted at p
void munch_list( morsel* p ) {
   tbb::task_group g;
   while( p ) {
      // Call munch on an item
      g.run( [=]{p->munch();} );
      // Advance to the next item
      p = p->next;
   }
   // Wait for all tasks to complete
   g.wait();
}
```

LISTING C.2

Using task_group.

convenient interfaces. Tasks can be spawned explicitly, or implicitly when all of their predecessor tasks complete. See the discussion of Listing 8.13 on pages 236–237 for how to use it.

C.9.1 empty_task

A `tbb::empty_task` is a `task` that does nothing. It is sometimes used for synchronization purposes, as in Listing 8.13 on pages 236–237.

C.10 atomic

Atomic objects have update operations that appear to happen instantaneously, as a single indivisible task. They are often used for lock-free synchronization. Atomic objects can be declared as instances of the class template `tbb::atomic<`T`>`, where T is an integral, enum, or pointer type. Listing C.3 shows an example use case.

```
float array[N];
tbb::atomic<int> count;

void append( float value ) {
   array[count++] = value;
}
```

LISTING C.3

Example of using atomic<int> as a counter.

If m threads execute `count++` at the same time, and its initial value is k, each thread will get a distinct result value drawn from the set $k, k+1, \ldots, k+m-1$, and afterward `count` will have value $k+m$. This, is true even if the threads do this simultaneously. Thus, despite the lack of mutexes, the code correctly appends items to the array.

In the example it is critical to use the value returned by `count++` and not reread `count`, because another thread might intervene and cause the reread value to be different than the result of `count++`.

Here is a description of the atomic operations supported by a variable `x` declared as a `tbb::atomic<X>`:

- *read, write*: Reads and writes on `x` are atomic. This property is not always true of non-atomic types. For example, on hardware with a natural word size of 32 bits, often reads and writes of 64-bit values are not atomic, even if executed by a single instruction.
- *fetch-and-add*: The operations `x+=k`, `x-=k`, `++x`, `x++`, `--x`, and `x--` have the usual meaning, but atomically update `x`. The expression `x.fetch_and_add(k)` is equivalent to `(x+=k)-k`.
- *exchange*: The operation `x.fetch_and_store(y)` atomically performs `x=y` and returns the previous value of `x`.

```
// Node in a linked list
struct node {
   float data;
   node* next;
};

// Root of a linked list
tbb::atomic<node*> root;

// Atomically prepend node a to the list
void add_to_list( node* a ) {
   for(;;) {
      // Take snapshot of root
      node* b = root;
      // Use the snapshot as link for a
      a->next = b;
      // Update root with a if root is still equal to b
      if( root.compare_and_swap(a,b)==b ) break;
      // Otherwise start over and try again
   }
}

// Atomically grab pointer to the entire list and reset root to NULL
node* grab_list() {
   return root.fetch_and_store((node*)NULL);
}
```

LISTING C.4

Using atomic operations on a list.

- *compare-and-swap*: The operation `x.compare_and_swap(y,z)` atomically performs `if(x== z) x=y`, and returns the original value of `x`. The operation is said to succeed if the assignment happens. Code can check for success by testing whether the return value equals `z`.

Listing C.4 shows uses of compare-and-swap and exchange to manipulate a linked list.

Doing more complicated list operations atomically is beyond the scope of this appendix.

In particular, implementing *pop* with a compare-and-swap loop scheme similar to the one in `add_to_list` requires special care to avoid a hazard called the *ABA problem* [Mic04]. The code shown has a benign form of the ABA problem, which happens when:

1. A thread executes `node* b=a`, and `a` was NULL.
2. Another threads executes `add_to_list` and `grab_list`.
3. The thread in step 1 executes `root.compare_and_swap(a,b)`. The compare-and-swap sees that $a == NULL$ and succeeds, just as if no other thread intervened.

The point is that a successful compare-and-swap does *not* mean that no thread intervened. Here, there is no harm done because as long as `root==a->next` when the compare-and-swap succeeds, the resulting list is correct. But, in other operations on linked structures, the effects can corrupt the structure or even cause invalid memory operations on freed memory.

Compare-and-swap loops also require care if there might be heavy contention. If P threads execute a compare-and-swap loop to update a location, $P-1$ threads will fail and have to try again. Then $P-2$ threads will fail, and so forth. The net burden is $\Theta(P^2)$ attempts and corresponding memory traffic, which can saturate the memory interconnect. One way to avoid the problem is *exponential backoff*—wait after each compare-and-swap fails, and double the wait after each subsequent failure.

C.11 enumerable_thread_specific

An object *e* of type `enumerable_thread_specific<T>` has a separate instance (or "local view") of `T` for each thread that accesses it. The expression *e*`.local()` returns a reference to the local view for the calling thread. Thus, multiple threads can operate on a `enumerable_thread_specific` without locking. The expression *e*`.combine(`*combine*`)` returns a reduction over the local view. See Section 11.3 for more details on how to use `enumerable_thread_specific`.

C.12 NOTES ON C++11

Though TBB works fine with C++98, it is simpler to use with C++11. In particular, C++11 introduces **lambda expressions** (Section D.2) and `auto` declarations (Section D.1) that simplify use of TBB and other template libraries. Lambda expressions are already implemented in the latest versions of major C++ compilers. We strongly recommend using them to teach, learn, and use TBB, because once you get past the novelty, they make TBB code easier to write and easier to read.

Additionally, TBB implements most of some C++11 features related to threading, thus providing an immediate migration path for taking advantage of these features even before they are implemented by C++ compilers. This path is further simplified by the way that TBB's injection of these features into namespace `std` is optional.

These features are:

- `std::mutex`: A mutex with a superset of the C++11 interface. The superset includes TBB's interface for mutexes.
- `std::lock_guard`: C++11 support for exception-safe scoped locking.
- `std::thread`: A way to create a thread and wait for it to complete. Sometimes threads really are a better solution than tasks, particularly if the "work" must be preemptively scheduled or mostly involves waiting for something to happen. Also, note that threads provide mandatory parallelism, which may be important when interacting with the outside world or in a user interface. Tasks provide optional parallelism, which is better for efficient computation.
- `std::condition_variable`: A way to wait until the state protected by a mutex meets a condition.

The parts of the C++11 interface not implemented in TBB are those that involve time intervals, since those would have involved implementing the C++11 time facilities. However, TBB does have equivalents to this functionality, based on TBB's existing `tick_count` interface for time.

A condition variable solves the problem of letting a thread wait until a state protected by a mutex meets a condition. It is used when threads need to wait for some other thread to update some state protected by a mutex. The waiting thread(s) acquire the mutex, check the state, and decide whether to wait. They wait on an associated condition variable. The `wait` member function atomically releases the mutex and starts the wait. Another thread acquires mutex associated with the condition, modifies the state protected by the mutex, and then signals one or all of the waiter(s) when it is done. Once the mutex is released, the waiters reacquire the mutex and can recheck the state to see if they can proceed or need to continue waiting.

Condition variables should be the method of choice to have a thread wait until a condition changes. TBB makes this method of choice portable to more operating systems.

C.13 HISTORY

The development of TBB was done at Intel and with the involvement of one of the authors of this book, Arch Robison. We can therefore recount the history of TBB from a personal perspective.

TBB was first available as a commercial library from Intel in the summer of 2006, not long after Intel shipped its first dual-core processors. It provided a much needed comprehensive answer to the question, "What must be fixed or added to C++ for parallel programming?" TBB's key programming abstractions for parallelism focused on logical specification of parallelism via algorithm templates. By also including a task-stealing scheduler, a thread-aware memory allocator, portable mutexes, global timestamps, and concurrent containers, TBB provided what was needed to program for parallelism in C++. The first release was primarily focused on strict **fork–join** or loop-type **data parallelism**.

The success of Intel TBB would, however, have been limited if it had remained a proprietary solution. Even during the release of version 1.0, Intel was in discussions with early customers on the future direction of TBB in both features and licensing.

Watching and listening to early adopters, such as Autodesk Maya, highlighted that much of the value of TBB was not only for data parallelism but also for more general parallelism using tasks, pipelines, scalable memory allocation, and lower-level constructs like synchronization primitives. Intel also received encouragement to make TBB portable by creating and supporting it via an open source project.

This customer feedback and encouragement led, only a year later, to version 2.0, which included a GPL v2 with the runtime exception version of both the source and binaries, as well as maintaining the availability of non-GPL binaries. Intel's customers had said that this would maximize adoption, and the results have definitely shown they were right.

Intel increased the staffing on TBB, worked proactively to build a community to support the project, and continued to innovate with new usage models and features over the next few years. We have been amazed and humbled by the response of such users as Adobe Systems, Avid, Epic Games, Dream-Works, and many others, along with that of other community members. TBB now has a very large user community and has had contributions that have led to Intel TBB being ported to many operating systems, platforms, and processors. We appreciate Intel's willingness to let us prove that an open source project initiated by Intel, yet supporting non-x86 processors, not only made sense—but would be very popular with developers. We've definitely proven that!

Through the involvement of customers and community, TBB has grown to be the most feature-rich and comprehensive solution for parallel application development available today. It has also become the most popular!

The TBB project was grown by a steady addition of ports to a wide variety of machines and operating systems and the addition of numerous new features that have added to the applicability and power of TBB.

TBB was one of the inspirations for Microsoft's Task Parallel Library (TPL) for .NET and Microsoft's Parallel Patterns Library (PPL) for C++. Intel and Microsoft have worked jointly to specify and implement a common subset of functionality shared by TBB and Microsoft's Parallel Patterns Library (PPL). In some cases, Intel and Microsoft have exchanged implementations and tests to ensure compatibility. An appendix of *The TBB Reference Manual* summarizes the common subset.

The most recent version of TBB, version 4.0, adds a powerful capability for expressing parallelism as data flowing through a graph. Use of TBB continues to grow, and the open source project enjoys serious support from Intel and others.

The Intel Cilk Plus project complements TBB by supplying C interfaces, simpler syntax, better opportunity for compiler optimization, and data parallel operations that lead to effective vectorization. None of these would be possible without direct compiler support. Intel briefly considered calling Cilk Plus simply "compiled TBB." While this conveyed the desire to extend TBB for the objectives mentioned, it proved complicated to explain the name so the name Cilk Plus was introduced. The full interoperability between TBB and Cilk Plus increases the number of options for software developers without adding complications. Like TBB, Intel has open sourced Cilk Plus to help encourage adoption and contribution to the project. TBB and Cilk Plus are sister projects at Intel.

C.14 SUMMARY

Intel Threading Building Blocks is a widely used and highly portable template library that provides a comprehensive set of solutions to programs using tasks in C++. It also provides a set of supporting functionality that can be used with or without the tasking infrastructure, such as concurrency-safe STL-compatible data structures, memory allocation, and portable atomics. Although we focus on tasks in this book due to their increased machine independence, safety, and scalability over threads, TBB also implements a significant subset of the C++11 standard's thread support, including platform-independent mutexes and condition variables. Much more information is available at `http://threadingbuildingblocks.org`.

APPENDIX

Glossary

E

The specialized vocabulary used in this book is defined here. In some cases, where existing terminology is ambiguous, we have given all meanings but note which meaning we primarily use in this book.

absolute speedup: Speedup in which the best parallel solution to a problem is compared to the best serial solution to the same problem, even if they use different algorithms. See ***relative speedup***.

access controls: Any mechanism to regulate access to something, but for parallel programs this term generally applies to shared memory. The term is sometimes extended to I/O devices as well. For parallel programming, the objective is generally to provide deterministic results by preventing an object from being modified by multiple tasks simultaneously. Most often this is referred to as ***mutual exclusion***, which includes locks, mutexes, atomic operations, and ***transactional memory*** models. This may also require some control on reading access to prevent viewing of an object in a partially modified state.

actual parallelism: The number of physical ***workers*** available to execute a parallel program.

algorithmic skeleton: Synonym for ***pattern***, specifically the subclass of patterns having to do with algorithms.

algorithm strategy pattern: A class of patterns that emphasize the parallelization of the internal workings of algorithms.

aliasing: Refers to when two distinct program identifiers or expressions refer to overlapping memory locations. For example, if two pointers `p` and `q` point to the same location, then `p[0]` and `q[0]` are said to alias each other. The potential for aliasing can severely restrict a compiler's ability to optimize a program, even when there is no actual aliasing.

Amdahl's Law: Speedup is limited by the non-***parallelizable*** serial portion of the work. Compare with other attempts to characterize the bounds of parallelism: ***span complexity*** and ***Gustafson–Barsis' Law***. See Section 2.5.4.

application binary interface (ABI): A set of binary entry points corresponding to an ***application programming interface***. Fixed ABIs are useful to allow relinking to different implementations of a library module.

application programming interface (API): An interface (set of function calls, operators, variables, and/or classes) by which an application developer uses a module. The implementation details of a module are ideally hidden from the application developer and the functionality is only defined through the API.

arithmetic intensity: The ratio of computational (typically arithmetic) operations to communication, where communication includes memory operations. Comparing this ratio for an algorithm with the hardware's ratio gives a hint of whether computation or communication will be the limiting resource. See Section 10.3.

array operations: See ***vector operations***.

array processors: See ***vector processor***.

array-of-structures (AoS): A data layout for collections of heterogeneous data where all the data for each element is stored in adjacent physical locations, even if the data are of different types. Compare with ***structure-of-arrays***.

associative cache: A cache organization where copies of data in main memory can be stored anywhere in the cache.

associative operation: An operation $\otimes$ is associative if $(a \otimes b) \otimes c = a \otimes (b \otimes c)$. Modular integer arithmetic is associative. Real addition is associative, but floating-point addition is not. However, sometimes the roundoff differences from reassociating floating-point addition are small enough to be ignored, in which case floating-point operations can be considered approximately associative.

asymptotic complexity: Algebraic limit on behavior, including time and space but also ratios such as *speedup and efficiency*. See *big O notation, big Omega notation*, and *big Theta notation*.

asymptotic efficiency: An *asymptotic complexity* measure for *efficiency*.

asymptotic speedup: An *asymptotic complexity* measure for *speedup*.

atomic operation: An operation guaranteed to appear as if it occurred indivisibly without interference from other threads. For example, a processor might provide a memory increment operation. This operation needs to read a value from memory, increment it, and write it back to memory. An atomic increment guarantees that the final memory value is the same as would have occurred if no other operations on that memory location were allowed to happen between the read and the write. See Section C.10, and ***lock*** and ***mutual exclusion***.

atomic scatter pattern: A ***non-deterministic*** data pattern in which multiple writers to a single storage location result in exactly one value being written and all others being discarded. The value written is chosen non-deterministically from the multiple sources. The only guarantee is that the resulting value in the target memory locations will be one of the values being written by at least one of the writers. See Section 6.2.

attached co-processor: A separate processor, often on an add-in card (such as a PCIe card), usually with its own physical memory, which may or may not be in a separate address space from the host processor. Often also known as an accelerator (although it may only accelerate specific workloads).

auto-vectorization: Automatically generating ***vectorized*** code from programs expressed using serial programming languages.

autotuning: The process of automatically adjusting parameters in parameterized code in order to achieve optimal performance.

available parallelism: See ***potential parallelism***.

bandwidth: The rate at which information is transferred, either from memory or over a communications channel. This term is used when the process being measured can be given a frequency-domain interpretation. When applied to computation, it can be seen as being equivalent to ***throughput***.

barrier: When a computation is broken into phases, it is often necessary to ensure that all threads complete all the work in one phase before any thread moves onto another phase. A barrier is a form of ***synchronization*** that ensures this. Threads arriving at a barrier wait there until the last thread arrives, then all threads continue. A barrier can be implemented using an ***atomic operation***. For example, all threads might try to increment a shared variable, then ***block*** if the value of that variable does not equal the number of threads that need to synchronize at the barrier. The last thread to arrive can then reset the barrier to zero and release all the blocked threads.

big O notation: Complexity notation that denotes an upper bound; written as $O(f(n))$. Big O notation is useful for classification of algorithm efficiency. In particular, big O notation is used to classify algorithms based on how they respond to changes in their input set in terms of processing time or other characteristics of interest.

For instance, a bubble sort routine may be described as taking $O(n^2)$ time because the time to run a bubble sort routine is proportional to the square of the size of the data set to sort. Since big O notation is about asymptotic growth, it may neglect significant constant factors. A pair of algorithms with running times of $n^2 + 100n + 10^{19}$ and $5n^2 + n + 2$, respectively, are both generally described as $O(n^2)$, despite significant differences in performance for most values of n.

For characterizing the suitability of an algorithm for parallel execution, ***big O*** analysis applies to both the ***work complexity*** and the ***span complexity***, but typically ***big Theta notation*** is preferred. See Section 2.5.7.

big Omega notation: Complexity notation that denotes a lower bound; written as $\Omega(f(N))$. See Section 2.5.7.

big Theta notation: Complexity notation that denotes an upper and a lower bound; written as $\Theta(f(N))$. See Section 2.5.7.

binning: The process of subdividing labeled data in a collection into separate sub-collections, each with a unique label. See ***bin pattern***.

bin pattern: A generalized version of the ***split pattern***, which is in turn a generalization of the ***pack pattern***, the bin pattern takes as input a collection of data and a collection of labels to go with every element of that collection, and reorganizes the data into a category (a bin) for every unique label in the input. The determinisitic version of this pattern is stable, in that it preserves the original order of the input collection. One major application of this pattern is in radix sort. It can also be used to implement the ***category reduction pattern***. See Section 6.4.

BLAS: The Basic Linear Algebra Subprograms are routines that provide standard building blocks for basic vector and matrix operations. The Level 1 BLAS perform scalar, vector, and vector–vector operations; the Level 2 BLAS perform matrix–vector operations; and the Level 3 BLAS perform matrix–matrix operations. Because the BLAS are efficient, portable, and widely available, they are commonly used in the development of high-quality linear algebra software (LAPACK, for example). A sophisticated and generic implementation of BLAS has been maintained for decades at `http://netlib.org/blas`. Vendor-specific implementations of BLAS are common, including the Intel Math Kernel Library (MKL), which is a highly efficient version of BLAS and other standard routines for Intel architecture.

block: Block can be used in two senses: (1) a state in which a thread is unable to proceed while it waits for some synchronization event, or (2) a region of memory. The second meaning is also used in the sense of dividing a loop into a set of parallel tasks of a suitable ***granularity***. To avoid confusion in this book, the term ***tile*** is generally used for the second meaning, and likewise the term ***tiling*** is preferred over "blocking."

branch and bound pattern: A ***non-deterministic*** pattern designated to find one satisfactory answer when many may be possible. Branch refers to using concurrency, and bound refers to limiting the computation in some manner—for example, by using an upper bound (perhaps the best result found so far). This pattern is often used to implement search, where it is highly effective. See Section 3.7.1.

burdened span: The ***span*** augmented with overhead costs.

by reference: A parameter to a function that acts exactly as if it were the original location passed to the function.

by value: A parameter to a function that is a copy of the original value passed to the function.

cache: A part of memory system that stores copies of data temporarily in a fast memory so that future uses for that data can be handled more quickly than if the request had to be fetched again from a more distant storage. Caches are generally automatic and are designed to enhance programs with ***temporal locality*** and/or ***spatial locality***. Caching systems in modern computers are usually multileveled.

cache coherence: A mechanism for keeping multiple copies of the same data in different caches consistent.

cache conflict: When multiple locations in memory are mapped to the same location in a cache only a subset of them can be kept in cache.

cache fusion: An optimization for a sequence of /vector operations/ where the vector operations are broken into ***tiles*** and the entire sequence executed on each tile, so that the intermediate values can be kept in cache.

cache line: The units in which data retrieved and held by a ***cache***; in order to exploit spatial locality, they are generally larger than a word. The general trend is for increasing cache line sizes, which are generally large enough to hold at least two double-precision floating-point numbers, but unlikely to hold more than eight on any current design. Larger cache lines allow for more efficient bulk transfers from main memory but worsen certain issues, including ***false sharing*** which generally degrades performance.

cache-oblivious programming: Refers to designing an algorithm to have good cache behavior without knowing the size or design of the cache system in advance. This is usually accomplished by using recursive patterns of data locality so that locality is present at all scales. See Section 8.8, as well as [ABF05] and [Vit08].

cancellation: The ability to stop a running (or ready to run) task from another task. Used in the speculative selection pattern discussed in Section 3.6.3.

category reduction pattern: A combination of search and segmented reduction, this is the form of reduction used in the map–reduce programming model. Each input has a label, and reduction occurs only between elements with the same label. The output is a set of reduction results for each unique label. See Section 3.6.8.

circuit complexity: See ***span complexity***.

closures: Objects that consist of a function definition and a copy of the environment (that is, the values of all variables referenced by the function) in effect at the time and visible from the scope in which the function was defined. See ***lambda function*** and Section 3.4.4.

cloud: A set of computers, typically maintained in a data center, that can be allocated dynamically and accessed remotely. Unlike a ***cluster***, cloud computers are typically managed by a third party and may host multiple applications from different, unrelated users.

cluster: A set of computers with distributed memory communicating over a high-speed interconnect. The individual computers are often called ***nodes***.

codec: An abbreviation for coder–decoder, a module that implements a data compression and decompression algorithm in order to reduce memory storage or communication bandwidth. For example, codecs that compress to/from MPEG4 are common for video.

code fusion: An optimization for a sequence of ***vector operations*** that combines the operations into a single ***elemental function***.

coherent masks: When the SPMD programming model is emulated on SIMD machines using ***masking***, the situation where the masks contain all 0's or all 1's.

collective operation: An operation, such as a ***reduction*** or a ***scan***, that acts on a collection of data as a whole. See Chapter 5.

collision: In the ***scatter pattern***, or when using random writes from parallel tasks, a collision occurs when two items try to write to the same location. The result is typically ***non-deterministic*** since it depends on the timing of the writes. In the worst case, a collision results in garbage being written to the location if the writes are not ***atomic*** and are not protected with ***locks***. See Sections 3.5.5 and 6.2.

combiner operation: The (ideally) associative and (possibly) commutative operation used in the definition of ***collective operations*** such as ***reduction*** and ***scan***. See Chapter 5.

communication: Any exchange of data or ***synchronization*** between software tasks or threads. Understanding that communication costs are often a limiting factor in scaling is a critical concept for parallel programming.

communication avoiding algorithm: An algorithm that avoids communication, even if it results in additional or redundant computation.

commutative operation: A commutative operation $\oplus$ satisfies the equation $a \oplus b = b \oplus a$ for all a and b in its domain. Some techniques for vectorizing reductions require commutativity.

composability: The ability to use two components with each other. Can refer to system features, programming models, or software components. See Section 1.5.4.

concurrent: Logically happening simultaneously. Two tasks that are both logically active at some point in time are considered to be concurrent. Contrast with ***parallel***.

continuation: The state necessary to continue a program from a certain logical location in that program. A well-known example is the statement following a subroutine call, which will be where a program continues after a subroutine finishes (returns). The continuation is more than just the location; it also includes the state of data, variable declarations, and so forth at that point.

continuation passing style: A style of programming in which the ***continuations*** of operations are explicitly created and managed.

control dependency: A ***dependency*** between two tasks where whether or not a task executes depends on the result computed by another task.

convergent memory access: When memory accesses in adjacent ***SIMD lanes*** access adjacent memory locations.

cooperative scheduling: A thread scheduling system that allows thread to switch tasks only at predictable switch points.

core: A separate subprocessor on a multicore processor. A core should be able to support (at least one) separate and divergent flow of control from other cores on the same processor. Note that there is some inconsistency in the use of this term. For example, some graphic processor vendors use the term as well for SIMD ***lanes*** supporting ***fibers***. However, the separate flows of control in fibers are simulated with masking on these devices, so there is a performance penalty for divergence. We will restrict the use of the term ***core*** to the case where control flow divergence can be done without penalty.

critical path: The longest chain of ***tasks*** ordered by ***dependencies*** in a program.

DAG: See ***directed acyclic graph***.

data dependency: A ***dependency*** between two tasks where one task requires as input data the output of another task.

data locality: See ***locality.***

data parallelism: An attempt to an approach to parallelism that is more oriented around data rather than tasks. However, in reality, successful strategies in parallel algorithm development tend to focus on exploiting the parallelism in data, because data decomposition (generating tasks for different units of data) scales, but functional decomposition (generation of hetereogeneous tasks for different functions) does not. See ***Amdahl's Law***, ***Gustafson–Barsis' Law***, and Section 2.2.

deadlock: A programming error that occurs when at least two tasks wait for each other and each will not resume until the other task proceeds. This happens easily when code requires locking multiple mutexes; for example, each task can be holding a mutex required by the other task. See Section 2.6.3.

dependencies: A relationship among tasks that results in an ordering constraint. See ***data dependency*** and ***control dependency***.

depth: See ***span complexity***.

deque: A double-ended queue.

design pattern: A general term for ***pattern*** that includes not only ***algorithmic strategy patterns*** but also patterns related to overall code organization.

deterministic: A deterministic algorithm is an algorithm that behaves predictably. Given a particular input, a deterministic algorithm will always produce the same output. The definition of what is the "same" may be important due to limited precision in mathematical operations and the likelihood that optimizations including ***parallelization*** will rearrange the order of operations. These are often referred to as "rounding" differences, which result when the order of mathematical operations to compute answers differs between the original program and the final concurrent program. Concurrency is not the only factor that can lead to ***non-deterministic*** algorithms but in practice it is often the cause. Use of programming models with sequential semantics and eliminating data races with proper access controls will generally eliminate non-determinism other than the "rounding" differences.

directed acyclic graph: A graph that defines a partial order so that nodes can be sorted into a linear sequence with references only going in one direction. A directed acyclic graph has, as its name suggests, directed edges and no cycles.

direct memory access (DMA): The ability of one processing unit to access another processing unit's memory without the involvement of the other processing unit.

direct-mapped cache: A cache in which every location in memory can be stored in only one location in the cache, typically using a modular function of the address.

distributed memory: Memory which is located in multiple physical locations. Accessing data from more remote locations typically has higher ***latency*** and possibly lower ***bandwidth*** than accessing local memory.

distributed memory: Memory that is physically located in separate computers. An indirect interface, such as message passing, is required to access memory on remote computers, while local memory can be accessed directly. Distributed memory is typically supported by ***clusters***, which, for purposes of this definition, we are considering to be a collection of computers. Since the memory on ***attached co-processors*** also cannot typically be addressed directly from the host, it can be considered, for functional purposes, to be a form of distributed memory.

divergent memory access: When memory accesses in adjacent SIMD lanes access non-adjacent memory locations. See ***convergent memory access***.

divide-and-conquer pattern: Recursive decomposition of a problem. Can often be parallelized with the ***fork–join*** parallel pattern. See Section 8.1.

domain-specific language (DSL): A language with specialized features suitable for a specific application domain, along with (typically) some restrictions to make optimization for that domain easier. For instance, an image processing language might support direct specification of the ***stencil pattern*** but restrict the use of general pointers. Domain-specific languages are often ***embedded languages***, in which case they are called *embedded domain-specific languages*, or EDSLs.

dwarf: A workload is which typical of some class of workloads. Sometimes used as a synonym for ***pattern***.

efficiency: Efficiency measures the return on investment in using additional hardware to operate in parallel. See Section 2.5.2.

elemental function: A function used in a ***map pattern***. An elemental function syntactically is defined as acting on single item inputs, but in fact is applied in parallel to all the elements of a collection. An elemental function can be vectorized by replicating the computation it specifies across multiple ***SIMD*** lanes. See Sections 4.1 and B.10.

embarrassing parallelism: Refers to an algorithm that can be decomposed into a large number of independent tasks with little or no synchronization or communication required. See ***map pattern***.

embedded language: A programming system whose syntax is supported using another language; for example, ArBB supports an embedded interface in C++. The computations specified using this interface are not, however, performed by C++. Instead, ArBB supports a set of types and operations in C++. Sequences of these operations can be recorded by ArBB and are then dynamically recompiled to machine language. See Section B.10.

expand pattern: A pattern in which each element of a ***map pattern*** can output zero or more data elements, which are then assembled into a single (possibly segmented) array. Related to the ***pack pattern***. See Sections 3.6.7 and 6.4.

false sharing: Two separate tasks in two separate cores may write to separate locations in memory, but if those memory locations happened to be allocated in the same cache line, the cache coherence hardware will attempt to keep the cache lines coherent, resulting in extra interprocessor communication and reduced performance, even though the tasks are not actually sharing data. See Section 2.4.2.

fiber: A very lightweight unit of parallelism that (conceptually) has its own flow of control but is mapped onto a single lane of a SIMD processor. Divergent control flow between fibers on a single SIMD processor is simulated by masking updates to registers and memory. See Section 2.3. A masked implementation has implications for performance. In particular, divergent control flow reduces lane utilization. There may also be limitations on control flow; for example, GOTO may not be supported, only nested control flow. Note that the term *fiber* is not universally accepted. In particular, on GPUs, fibers are often called ***threads*** and what we call threads are called *work groups* in OpenCL.

Fibonacci numbers: The Fibonacci numbers are defined by linear recurrence relationship and suffer from overuse in computer science as examples of recursion as a result. Fibonacci numbers are defined as $F(0) = 0$ and $F(1) = 1$ plus the relationship defined by $F(N) = F(N-1) + F(N-2)$.

fine-grain locking: Locks that are used to protect parts of a larger data structure from race conditions. Such locks avoid locking the entirety of a large data structure during parallel accesses.

fine scale: A level of parallelism with very small units of parallel work. Reduction of overhead is very important for fine-scale parallelism to be effective, since otherwise the overhead will dominate the computation.

Flynn's characterization: A classic categorization of parallel processors by Flynn [Fly72] based on whether they have multiple flows of control or multiple streams of data. See Section 2.4.3.

fold: A collective operation in which every output is a function of all previous outputs and all inputs up to the current output point. A fold is based on a ***successor function*** that computes a new output value and a new state for the fold for each new input value. A ***scan*** is a special, parallelizable case of a fold where the successor function is associative.

fork: The creation of a new thread or task. The original thread or task continues in parallel with the forked thread or task. See ***spawn***.

fork–join pattern: A pattern of computation in which new (potential) parallel flows of control are created/split with ***forks*** and terminated/merged with ***joins***. See Sections 3.3.1 and 8.1.

fork point: A point in the code where a ***fork*** takes place.

fully associative cache: See ***associative cache***.

functional decomposition: An approach to parallelization of existing serial code where modules are run on different threads. This approach does not give more than a constant factor of speedup at best since the number of modules in a program is fixed.

functional unit: A hardware processing element that can do a simple operation, such as a single arithmetic operation.

functor: A class which supports a function-call interface. Unlike functions in C and C++ however, functors can also hold state and can support additional interfaces to modify that state. See ***lambda functions***.

fusion: An optimization in which two or more things with similar forms are combined. See ***loop fusion, cache fusion***, and ***code fusion***.

future: An approach to asynchronous computing in which a computation is specified but does not necessarily begin immediately. Instead, construction of a future returns an object which can be used to query the status of the computation or wait for its completion.

future-proofed: A computer program written in a manner so it will survive future computer architecture changes without significant changes to the program itself being necessary. Generally, the more abstract a programming method is, the more future-proof that program is. Lower-level programming methods that in some way mirror computer architectural details will be less able to survive the future without change. Writing in an abstract, more future-proof fashion may involve tradeoffs in efficiency, however.

gather pattern: A set of parallel random reads from memory. A gather takes a collection of addresses and an input collection and returns a collection of data drawn from the input collection at the given locations. Gathers are equivalent to random reads inside a ***map pattern***. See Sections 3.5.4 and 6.1.

geometric decomposition pattern: A pattern that decomposes the computational domain for an algorithm into a set of possibly overlapping subdomains. A special case is the ***partition pattern***, which is when the subdomains do not overlap. See Sections 3.5.3 and 6.6.

GPU: A graphics processing unit is an attached graphics processor originally specialized for graphics computations. GPUs are able to support arbitrary computation, but they are specialized for massively parallel, fine-grained computations. They typically use multithreading, ***hyperthreading***, and **fibers** and make extensive use of ***latency hiding***. They are typically able to maintain the state for many more threads in memory than CPUs, but each thread can have less total state.

grain: A unit of work to be run serially. See ***granularity***.

grain size: The amount of work in a ***grain***.

granularity: The amount of decomposition applied to the ***parallelization*** of an algorithm, and the **grain size** of that decomposition. If the granularity is too coarse, there are not enough parallel tasks to effectively make use of all parallel hardware units and hide latency. If the granularity is too fine, there are too many parallel tasks and overhead may dominate the computation.

graphics accelerators: A processor specialized for graphics workloads, usually in support of real-time graphics APIs such as Direct3D and OpenGL. See ***GPU***.

graph rewriting: A computational pattern where nodes of a graph are matched against templates and substitutions made with other subgraphs. When applied to directed acyclic graphs (trees with sharing), this is known as *term graph rewriting* and is equivalent to the lambda calculus, except that it also explicitly represents sharing of memory. See Section 3.6.9.

greedy scheduling: A scheduling strategy in which no worker idles if there is work to be done.

grid: A distributed set of computers that can be allocated dynamically and accessed remotely. A grid is distinguished from a cloud in that a grid may be supported by multiple organizations and is usually more heterogeneous and physically distributed.

Gustafson–Barsis' Law: A different view on Amdahl's Law that factors in the fact that as problem sizes grow the serial portion of computations tend to shrink as a percentage of the total work to be done. Compare with other attempts to characterize the bounds of parallelism, such as ***Amdahl's Law*** and ***span complexity***. See Section 2.5.5.

halo: In the implementation of the ***stencil pattern*** on ***distributed memory*** a set of elements surrounding a ***partition*** that are replicated on different workers to allow each portion of the partition to be computed in parallel.

hardware thread: A hardware implementation of a task with a separate flow of control. Multiple hardware threads can be implemented using multiple cores, or they can run concurrently or simultaneously on one core in order to hide latency using methods such as ***hyperthreading*** of a processor core. See Sections 1.2 and 2.5.9.

heap allocation: An allocation mechanism that supports unstructured memory allocations of different sizes and at arbitrary times in the execution of a program. Compare with ***stack allocation***.

heterogeneous computer: A computer which supports multiple processors each with specialized capabilities or performance characteristics.

holders: A form of /hyperobject/ useful for managing temporary task-local storage.

host processor: The main control processor in a system, as opposed to any graphics processors or co-processors. The host processor is responsible for booting and running the operating system.

hyperobjects: A mechanism in Cilk Plus to support operations such as reduction that combine multiple objects. See Section B.7. For examples using hyperobjects, see Sections 5.3.5, 8.10, and 11.2.1.

hyperthreading: Multithreading on a single processor core. With hyperthreading, also called simultaneous multithreading, multiple ***hardware threads*** may run on one core and share resources, but some benefit is still obtained from parallelism or concurrency. For example, the processor may draw instructions from multiple hyperthreads to fill superscalar instruction slots, or the processor may switch between multiple hyperthreads in order to hide memory access latency. Typically each hyperthread has, at least, its own register file and program counter, so that switching between hyperthreads is relatively lightweight.

implementation pattern: A pattern that is specific to efficient implementation (usually of some other pattern) using specific hardware mechanisms.

instance: In a ***map pattern*** one invocation of an elemental function on one element of the map.

instruction-level parallelism (ILP) wall: The limits to automatic parallelism given by the amount of parallelism naturally available at the instruction level in serial programs.

intrinsics: Intrinsics appear to be functions in a language but are supported by the compiler directly. In the case of SSE or vector intrinsics, the intrinsic function may map directly to a small number, often one, of machine instructions which the compiler inserts without the overhead of a real function call. For a discussion of SSE intrinsics, see Section 5.3.3.

irregular parallelism: parallelism with disimiliar tasks with unpredictable dependencies.

iteration pattern: A serial pattern in which the same sequence of instructions is executed repeatedly and in sequence.

join: When multiple flows of control meet and a single flow continues onwards. Not to be confused with a ***barrier***, in which all the incoming flows continue onwards.

join point: A point in the code where a ***join*** takes place.

kernel: A general term for a small section of code that (1) executes a large amount of computation relative to other parts of the program (also known as a hotspot), and/or (2) is the key code sequence for an algorithm.

lambda expression: an expression that returns a ***lambda function***.

lambda function: A lambda function, for programmers, is an anonymous function. Long a staple of languages such as LISP, it was only recently supported for C++ per the C++11 standard. A lambda function enables a fragment of code to be passed to another function without having to write a separate named function or functor. This ability is particularly handy for using TBB. See Section D.2.

lane: An element of a SIMD register file and associated functional unit, considered as a unit of hardware for performing parallel computation. SIMD instructions execute computations across multiple lanes.

latency: The time it takes to complete a task—that is, the time between when the task begins and when it ends. Latency has units of time. The scale can be anywhere from nanoseconds to days. Lower latency is better in general. See Section 2.5.1.

latency hiding: Schedules computations on a processing element while other tasks using that core are waiting for long-latency operations to complete, such as memory or disk transfers. The latency is not actually hidden, since each task still takes the same time to complete, but more tasks can be completed in a given time since resources are shared more efficiently, so throughput is improved. See Section 2.5.9.

latent parallelism: See ***potential parallelism***.

linear speedup: Speedup in which the performance improves directly proportional to the physical processing resources available. Considered to be optimal.

Little's formula: A formula relating parallelism, concurrency, and latency.

livelock: A situation in which multiple workers are active, but are not doing useful work and are not making forward progress. See ***deadlock***.

load balancing: Distributing work across resources so that no resource idles while there is work to be done.

load imbalance: A situation where uneven sizes of tasks assigned to workers results in some workers finishing early and then idling while waiting for other workers to complete larger tasks. See ***load balancing***.

locality: Refers to either ***spatial locality*** or ***temporal locality***. Maintaining a high degree of locality of reference is a key to scaling. See Section 2.6.5.

lock: A mechanism for implementing ***mutual exclusion***. Before entering a mutual exclusion region, a thread must first try to acquire a lock on that region. If the lock has already been acquired by another thread, the current thread must ***block***, which it may do by either suspending operation or spinning. When the lock is released, then the current thread is free to acquire it. Locks can be implemented using ***atomic operations***, which are themselves a form of mutual exclusion on basic operations, implemented in hardware. See Section 2.6.2.

loop-carried dependencies: A dependency that exists between multipleiterations of an ***iteration pattern***.

loop fusion: An optimization where two loops with the compatible indexing executed in sequence can be combined into a single loop.

mandatory concurrency: See ***mandatory parallelism***.

mandatory parallelism: Parallelism that is semantically required for program correctness. See Section 9.6.

many-core processor: A ***multicore*** processor with so many cores that in practice we do not enumerate them; there are just "lots." The term has been generally used with processors with 32 or more cores, but there is no precise definition.

map pattern: Replicates a function that is applied to all elements of a collection, producing a new collection with the same shape as the input. The function being replicated is called an ***elemental function*** since it applies to the elements of an actual collection of input data. See Sections 3.3.2 and Chapter 4.

masking: A technique for emulating ***SPMD*** control flow on ***SIMD*** machines in which elements that are not active are prohibited from updating externally visible state.

megahertz era: A historical period of time during which processors doubled clock rates at a rate similar to the doubling of transistors in a design, roughly every 2 years. Such rapid rise in processor clock speeds ceased at just under 4 GHz (4,000 megahertz) in 2004. Designs shifted toward adding more cores, marking the shift to the ***multicore era***.

member function: A function associated with an object and which can access instance-specific object state.

memory fences: A synchronization mechanism which can ensure that memory operations before the fence are completed and are visible before memory operations after the fence.

memory hierarchy: See ***memory subsystem***.

memory subsystem: The portion of a computer system responsible for moving code and data between the main system memory and the computational units. The memory subsystem may include additional connections to I/O devices including graphics cards, disk drives, and network interfaces. A modern memory subsystem will generally have many levels, including some levels of caching both on and off the processor die. Coherent memory subsystems, which are used in most computers, provide for a single view of the contents of the main system memory despite temporary copies in caches and concurrency in the system. See Section 2.4.1.

memory wall: A limit to parallel scalability given by the fact that memory (and more generally, communication) ***bandwidth*** and in particular ***latency*** are not scaling at the same rate as computation.

merge scatter pattern: In a merge scatter, results that collide while implementing a ***scatter pattern*** are combined with an associative operator. The operator needs to be associative so the answer is the same regardless of the order in which elements are combined. We might also want to use this operator to combine scattered values with the previous contents of the target array. The merge scatter pattern can be used to implement histograms, for example. See Section 6.2.

metaprogramming: The use of one program to generate or manipulate another, or itself. See also ***template metaprogramming***.

method: See ***member function***.

MIC: The Intel Many Integrated Core architecture is designed for highly parallel workloads. The architecture emphasizes higher core counts on a single die, and simpler more efficient cores, than on a traditional CPU. A prototype with up to 32 cores and based on 45-nm process technology, known as Knight Ferry, was made available, but not sold, by Intel in 2010 and 2011. A product built on 22-nm process technology with more than 50 cores is expected in late 2012 or sometime in 2013.

MIMD: Multiple Instruction, Multiple Data, one of Flynn's classes of computer that supports multiple threads of control, each with its own data access. See ***SIMD*** and Section 2.4.3.

monoid: An ***associative operation*** that has an identity.

Moore's Law: Describes a long-term trend that the number of transistors that can be incorporated inexpensively on an integrated circuit chip doubles approximately every 2 years. It is named for Intel co-founder Gordon Moore, who described the trend in his 1965 paper in *Electronics Magazine*. This forecast of the pace of silicon technology has essentially described the basic business model for the semiconductor industry as well as being a driving force of technological and social change since the late 20th century.

motif: Sometimes used as a synonym for ***pattern***.

multicore: A processor with multiple subprocessors, each subprocessor (known as a ***core***) supporting at least one hardware thread.

multicore era: Time after which processor designs shifted away from rapidly rising clock rates and shifted toward adding more cores. This era began roughly in 2005.

multiple-processor systems: A system with two or more processors implemented on separate physical dies.

mutex: Short for **/*mutual exclusion*/**, and also used as a synonym for ***lock***.

mutual exclusion: A mechanism for protecting a set of data values so that while they are manipulated by one parallel thread they cannot be manipulated by another. See ***lock*** and ***transactional memory***.

nesting pattern: Refers to the ability to hierarchically compose other patterns. The nesting pattern simply means that all "tasks" in the pattern diagrams within this book are actually locations within which general code can be inserted. This code can in turn be composed of other patterns.

Network interface controller (NIC): A specialized communication processor.

node (in a cluster): A shared memory computer, often on a single board with multiple processors, that is connected with other nodes to form a ***cluster*** computer or supercomputer.

non-deterministic: Exhibiting a lack of deterministic behavior, so results can vary from run to run of an algorithm. See more in the definition for ***deterministic***.

non-uniform memory access (NUMA): A memory system in which certain banks of memory take longer to access than others, even though all the memory uses a single address space. See also ***distributed memory***.

objects: Objects are a language construct that associate data with the code to act on and manage that data. Multiple functions may be associated with an object and these functions are called the *methods* or *member functions* of that object. Objects are considered to be members of a class of objects, and classes can be arranged in a hierarchy in which subclasses inherit and extend the features of superclasses. The state of an object may or may not be directly accessible; in many cases, access to an object's state may only be permitted through its methods. See Section 3.4.5.

offload: Placing part of a computation on an ***attached device*** such as a GPU or co-processor.

online: An algorithm which can begin execution before all input data is read.

OpenCL: Open Computing Language, initiated by Apple Corporation, is now a standard defined by the Khronos group for graphics processors and ***attached co-processors***. However, OpenCL can also be used to specify parallel and vectorized computations on multicore host processors.

optional parallelism: Parallelism that is specified by a programming model but is not semantically necessary. Antonym is ***mandatory parallelism***.

over-decomposition: A parallel programming style where many more tasks are specified than there are physical workers for executing it. This can be beneficial for ***load balancing*** particularly in systems that support ***optional parallelism***.

over-subscription: More threads run on a system than it has physical workers, resulting in excessive overhead for switching between multiple threads or exceeding the number of threads that can be supported by the operating system. This can be avoided by using a programming model with ***optional parallelism***.

pack pattern: A data management pattern where certain elements of a collection are discarded and the remaining elements are placed in a contiguous sequence, maintaining the order of the original input. Related to the ***expand pattern***.

page: The granularity at which virtual to physical address mapping is done. Within a page, the mapping of virtual to physical memory addresses is continuous. See Section 2.4.1.

parallel: Physically happening simultaneously. Two tasks that are both actually doing work at some point in time are considered to be operating in parallel. When a distinction is made between ***concurrent*** and ***parallel***, the key is whether work can ever be done simultaneously. Multiplexing of a single processor core, by multitasking operating systems, has allowed concurrency for decades even when simultaneous execution was impossible because there was only one processing core.

parallel pattern: ***Patterns*** arising specifically in the specification of parallel applications. Examples of parallel patterns include the ***map pattern***, the ***reduction pattern***, the ***fork–join pattern***, and the ***partition pattern***.

parallel slack: The amount of "extra" parallelism available above the minimum necessary to use the parallel hardware resources. See Sections 2.4.2 and 2.5.6.

parallelism: Doing more than one thing at a time. Attempts to classify types of parallelism are numerous; read more about classifications of parallelism in Sections 2.2 and 2.3.

parallelization: The act of transforming code to enable simultaneous activities. The parallelization of a program allows at least parts of it to execute in parallel.

partition pattern: A pattern that decomposes the computational domain for an algorithm into a set of non-overlapping subdomains called ***tiles*** or ***blocks*** (although *tile* is the term preferred in this book). See the ***geometric decomposition pattern***, which is similar but allows overlap between subdomains. The partition pattern is a special case of the geometric decomposition pattern that does not allow overlap. See Section 6.6.

pattern: A recurring combination of data and task management, separate from any specific algorithm. Patterns are universal in that they apply to and can be used in any programming system. Patterns have also been called ***dwarfs***, ***motifs***, and algorithmic skeletons. Patterns are not necessarily tied to any particular hardware architecture or programming language or system. Examples of patterns include the ***sequence pattern*** and the ***object pattern***. See ***parallel pattern*** and Chapter 3.

PCIe bus: A peripheral bus supporting relatively high bandwidth and DMA, often used for attaching specialized co-processors such as ***GPUs*** and ***NICs***.

permutation scatter pattern: A form of the ***scatter pattern*** in which multiple writes to a single storage location are illegal. This form of scatter is deterministic, but can only be considered safe if collisions are checked for. See Section 6.2.

pipeline pattern: A set of data processing elements connected in series, generally so that the output of one element is the input of the next one. The elements of a pipeline are often executed concurrently. Describing many algorithms, including many signal processing problems, as pipelines is generally quite natural and lends itself to parallel execution. However, in order to scale beyond the number of pipeline stages, it is necessary to exploit parallelism within a single pipeline stage. See Sections 3.5.2, 9.2, 12.2, and C.6.

potential parallelism: At a given point of time, the number of parallel tasks that could be used by a parallel implementation of an algorithm, given sufficient hardware resources. Additional hardware resources above the potential parallelism in an algorithm are not usable. If the potential parallelism is larger than the physical parallelism, then the tasks will need to share physical resources by ***serialization***. Also known as ***latent parallelism*** and ***available parallelism***.

power wall: A limit to the practical clock rate of serial processors given by thermal dissipation and the non-linear relationship between power and switching speed.

pragma: A form of program markup used to give a hint to a compiler but not change the semantics of a program. Also called a "compiler directive."

precision: The detail in which a quantity is expressed. Lack of precision is the source of rounding errors in computation. The finite number of bits used to store a number requires some approximation of the true value. Errors accumulate when multiple computations are made to the data in operations such as reductions. Precision is measured in terms of the number of digits that contain meaningful data, known as significant digits. Since precision is most often considered in reference to floating-point numbers, significant digits in computer science have often been measured in bits (binary digits) because most floating-point arithmetic is done in radix-2. With the advent of IEEE-754-2008, radix-10 arithmetic is once again popular and precision of such data would be expressed in terms of decimal digits. See Section 5.1.4.

preemptive scheduling: A scheduling system that allows a thread to switch tasks at any time.

priority scatter pattern: A deterministic form of the ***scatter pattern*** in which an attempt to write multiple values in parallel to a single storage location results in one value (and only one) value being stored based on a priority function, while all other values are discarded. The unique priority given to each parallel write in a priority scatter can be assigned in such a way that the result is deterministic and equivalent to a serial implementation. See Section 6.2.

process: A application-level unit of parallel work. A process has its own thread of control and is managed by the operating system. Usually, unless special provisions are made for ***shared memory***, a process cannot access the memory of another process.

producer–consumer: A relationship in which the producer creates data that is passed to the consumer to utilize or further process. If data is not consumed exactly when it is produced, it must be buffered. Buffering introduces challenges of stalling the producer when the buffer is full, and stalling the consumer when the buffer is empty.

pure function: A function whose output depends only on its input, and that does not modify any other system state.

race condition: Non-deterministic behavior in a parallel program that is generally a programming error. A race condition occurs when concurrent tasks perform operations on the same memory location without proper synchronization and one of the memory operations is a write. Code with a race may operate correctly sometimes and fail other times. See Section 2.6.1.

recurrence pattern: A sequence defined by a recursive equation. In a recursive equation, one or more initial terms are given and each further term of the sequence is defined as a function of the preceding terms. Implementing recurrences with recursion is often inefficient since it tends to recompute elements of the recurrence unnecessarily. Recurrences also occur in loops with dependencies between iterations. In the case of a single loop, if the dependence is associative, it can be ***parallelized*** with the ***scan pattern***. If the dependence is inside a multidimensional loop nest, the entire nest can always be parallelized over $n-1$ dimensions using a hyperplane sweep, and it can also often be parallelized with the fork–join pattern. See Sections 3.3.6, 7.5, and 8.12.

recursion: The act of a function being re-entered while an instance of the function is still active in the same thread of execution. In the simplest and most common case, a function directly calls itself, although recursion can also occur between multiple functions. Recursion is supported by storing the state for the continuations of partially completed functions in dynamically allocated memory, such as on a stack, although if higher-order functions are supported a more complex memory allocation scheme may be required. Bounding the depth of recursion can be important to prevent excessive use of memory.

reduce: Apply operation to merge a collection of values to a single value. An example is summing a sequence of values. See ***reduction pattern***.

reducers: Hyperobjects that can implement ***reduce*** operations.

reduction pattern: The most basic ***collective*** pattern, a reduction combines all the elements in a collection into a single element using pairwise applications of a ***combiner operation***. In order to allow ***parallelization***, the combiner operation should be associative. In order to allow for efficient ***vectorization***, it is useful if the combiner operation is also commutative. Many useful reduction operations, such as maximum and (modular integer) addition, are both associative and commutative. Unfortunately, floating-point addition and multiplication are not, which can lead to potential ***non-determinism***. See Section 5.1.

reduction variable: A variable that appears in a loop for combining the results of many different loop iterations.

refactoring: Reorganizing code to make it better suited for some purpose, such as parallelization.

registers: Very fast but usually very limited on-core storage for intermediate results.

regular parallelism: A class of algorithms in which the tasks and data dependencies are arranged in a regular and predictable pattern.

relative speedup: Speedup in which a parallel solution to a problem is compared to a serialization of the same solution, that is, using the same algorithm. See ***absolute speedup***.

relaxed sequential semantics: See ***sequential semantics*** for an explanation.

response time: The time between when a request is made and when a response is received.

rotate pattern: A special case of the ***shift pattern*** that handles boundary conditions by moving data from the other side of the collection. See Section 6.1.2.

safety: A system property that automatically guards against certain classes of programmer errors, such as race conditions.

saturation: Saturation arithmetic has maximum and minimum values that are utilized when computation would logically arrive at higher or lower values if unbounded numerical representations were utilized. Saturation arithmetic is needed only because numerical representations on computer systems are almost always limited in precision and range. In floating-point arithmetic, the concept of positive and negative infinity as uniquely represented numbers in the floating-point format is utilized and is the default in instances of saturation. In integer arithmetic, wrap-around arithmetic is generally the default. Special instructions for saturation arithmetic are available in modern instruction sets (such as MMX), often originally motivated by graphics where the desire to make a graphical pixel brighter and brighter by increasing the value of a pixel was frustrated by a sudden dimming of the pixel due to wrap-around arithmetic. In an 8-bit unsigned number format, the addition of 254 with 9 will result in an answer of 7 in wrap-around or 255 in saturation arithmetic. Likewise, the subtraction of 11 from 7 would result in 252 in wrap-around vs. 0 in saturation arithmetic. Note, however, that saturation arithmetic for signed numbers is not associative.

scalability: A measure of the increase in performance as a function of the availability of more hardware to use in parallel. See Section 2.5.2.

scalable: An application is ***scalable*** if its performance increases when additional parallel hardware resources are added. See ***scalability***.

scalar promotion: When a scalar and a vector are combined using a vector operation, the scalar is automatically treated as a vector with all elements set to the same value.

scan pattern: Pattern arising from a one-dimensional recurrence relationship in the definition of a computation. This often arises as a loop-carried dependency where the computation of one iteration is dependent on the results of a prior iteration. Such loops are, surprisingly, still parallelizable if the dependency can be expressed as an associative operation. See Section 5.4.

scatter pattern: A set of input data and a set of indices is given, and each element of the input is written at the given location. Scatter can be considered the inverse of the ***gather pattern***. A collision in output occurs if the set of indices maps multiple input data to the same location. There are at least four ways to resolve such collisions: ***permutation scatter***, ***atomic scatter***, ***priority scatter***, and ***merge scatter***. See Section 3.5.5.

search pattern: A pattern that finds data that meets some criteria within a collection of data. See Section 3.6.5.

segmentation: A representation of a collection divided into non-uniform non-overlapping subdomains. Operations such as reduction and scan can be generalized to operate over the segments of a collection independently while still being perfectly load balanced. See Section 3.6.6.

selection pattern: A serial pattern in which one of two flows of control are chosen based on a Boolean control expression.

semantics: What a programming language construct does, as opposed to how it does it (pragmatics) or how it is expressed (syntax).

separating hyperplane: A plane that can be used to determine the sweep order for executing a multidimensional ***recurrence*** in parallel.

sequence pattern: The most fundamental serial pattern in which tasks are executed one after the other, with each task completing before the next one starts. See Section 3.2.1.

sequential bottlenecks: See ***serial bottlenecks***.

sequential consistency: Sequential consistency is a memory consistency model where every task in a concurrent system sees all memory writes (updates) happen in the exact same order, and a task's own writes occur in the order that the task specified. See Section 2.6.1.

sequential semantics: Refers to when a (parallel) program can be executed using a single thread of control as an ordinary sequential program without changing the semantics of the program. Parallel programming with sequential semantics has many advantages over programming in a manner that precludes serial execution and is therefore strongly encouraged. Such programs are considered easier to understand, easier to debug, more efficient on sequential machines, and better at supporting nested parallelism. Sequential semantics casts parallelism as an accelerator and not as mandatory for correctness. This means that one does not need a conceptual parallel model to understand or execute a program with sequential semantics. Examples of *mandatory parallelism* include producer–consumer relationships with bounded buffers (hence, the producer cannot necessarily be completely executed before the consumer because the producer can become blocked) and message passing (e.g., MPI) programs with cyclic message passing. Due to timing, precision, and other sources of inexactness, the results of a sequential execution may differ from the concurrent invocation of the same program. Sequential semantics solely means that any such variation is not due to the semantics of the program. The term "relaxed sequential semantics" is sometimes used to explicitly acknowledge the variations possible due to non-semantic differences in serial vs. concurrent executions. See Section 1.1 See ***serial semantics.***

serial: Neither concurrent nor parallel.

serial bottlenecks: A region of an otherwise parallel program that runs serially.

serial consistency: A parallel program that produces the same result as a specific serial ordering of its tasks.

serial elision: The serial elision of a Cilk Plus program is generated by erasing occurrences of the `cilk_spawn` and `cilk_sync` keywords and replacing `cilk_for` with `for`. Cilk Plus is a faithful extension of C/C++ in the sense that the serial elision of any Cilk Plus program is both a serial C/C++ program *and* a semantically valid implementation of the Cilk Plus program. The term *elision* arose from earlier versions of Cilk that lacked `cilk_for`, so eliding (omitting) the two other keywords sufficed. The term "C elision" is sometimes used, too, harking back to when Cilk was an extension of C but not C++. See Section B.4.

serial illusion: The apparent serial execution order of machine language instructions in a computer. In fact, hardware is naturally parallel, and many low-level optimizations and high-performance implementation techniques can reorder operations.

serial semantics: Same as ***sequential semantics***.

serial traps: A serial trap is a programming construct that semantically requires serial execution for proper results in general even though common cases may be overconstrained with regard to concurrency by such semantics. The term "trap" acknowledges how such constructs can easily escape attention as barriers to parallelism, in part because they are so common and were not intentionally designed to preclude parallelism. For instance, `for`, in the C language, has semantics that dictate the order of iterations by allowing an iteration to assume that all prior iterations have been executed. Many loops do not rely upon side-effects of prior iterations and would be natural candidates for parallel execution, but they require analysis in order for a system to determine that parallel execution would not violate the program semantics. Use of `cilk_for`, for instance, has no such serial semantics and therefore is not a serial trap. See examples in Section 1.3.3.

serialization: Refers to when the tasks in a potentially parallel algorithm are executed in a specific serial order, typically due to resource constraints. The opposite of ***parallelization.***

set associative cache: A cache architecture in which a particular location in main memory can be stored in a (small) number of different locations in cache.

shared address space: Even if units of parallel work do not share a physical memory, they may agree on conventions that allow a single unified set of addresses to be used. For example, one range of addresses could refer to memory on the host, while another range could refer to memory on a specific co-processor. The use of unified addresses simplifies memory management.

shared memory: Refers to when two units of parallel work can access data in the same location. Normally doing this safely requires ***synchronization***. The units of parallel work—***processes***, ***threads***, ***tasks***, and ***fibers***—can all share data this way, if the physical memory system allows it. However, processes do not share memory by default and special calls to the operating system are required to set it up.

shift pattern: A special case of the ***gather pattern*** that translates (that is, offsets the location of) data in a collection. There are a few variants based on how boundary conditions are handled. The basic pattern fills in a default value at boundaries, while the ***rotate pattern*** moves data from the other side of the collection. See Section 6.1.2.

SIMD: Single Instruction, Multiple Data, one of Flynn's classes of computer that supports a single operation over multiple data elements. See ***MIMD*** and Section 2.4.3.

simultaneous multithreading: A technique that supports the execution of multiple threads on a single core by drawing instructions from multiple threads and scheduling them in each superscalar instruction slot.

SIMT: Single Instruction, Multiple Threads, a variation on Flynn's characterizations that is really a collection of multiple SIMD processors, with control flow emulated on SIMD machines using a mechanism such as masking. See Section 2.4.3.

software thread: A software thread is a virtual hardware thread—in other words, a single flow of execution in software intended to map one for one to a hardware thread. An operating system typically enables many more software threads to exist than there are actual hardware threads by mapping software threads to hardware threads as necessary. See Section 2.3.

space complexity: A complexity measure for the amount of memory used by an algorithm as a function of problem size.

span: How long a program would take to execute on an idealized machine with an infinite number of processors. The ***span*** of an algorithm can also be seen as the critical path in its task dependency graph. See ***span complexity***.

span complexity: Span complexity is an asymptotic measure of complexity based on the ***span***. In the analysis of parallel algorithms and in particular in order to predict their ***scalability***, this measure is as important as ***work complexity***. Other synonyms for ***span complexity*** in the literature are ***step complexity***, ***depth***, or ***circuit complexity***. Compare with other attempts to characterize the bounds of parallelism: ***Amdahl's Law*** and ***Gustafson-Barsis' Law***. See Section 2.5.6.

spatial locality: Nearby when measured in terms of distance (in memory address). Compare with ***temporal locality***. Spatial locality refers to a program behavior where the use of one data element indicates that data nearby, often the next data element, will probably be used soon. Algorithms exhibiting good spatial locality in data usage can benefit from cache line structures and prefetching hardware, both common components in modern computers.

spawn: Generically, the creation of a new ***task***. In terms of Cilk Plus, `cilk_spawn` creates a spawn, but the new task created is actually the ***continuation*** and not the call that is the target of the spawn keyword. See ***fork***.

spawning block: The function, try block, or `cilk_for` body that contains the spawn. A sync (`cilk_sync`) waits only for spawns that have occurred in the same spawning block and have no effect on spawns done by other tasks or threads, nor those done prior to entering the current spawning block. A sync is always done, if there have been spawns, when exiting the enclosing spawning block.

speedup: Speedup is the ratio between the latency for solving a problem with one processing unit versus the latency for solving the same problem with multiple processing units in parallel. See Section 2.5.2.

split pattern: A generalized version of the ***pack pattern*** that takes an input collection and a set of Boolean labels to go with every element of that collection. It reorganizes the data so all the elements marked with `false` are at one end of the output collection (rather than discarding them as with the pack pattern), and all the elements marked with `true` are at the other end of the collection. The determinisitic version of this pattern is stable, in that it preserves the original order of the input collection in each output partition. One major application of this pattern is in base-2 radix sort. The ***bin pattern*** is a generalization to more than two categories. See Section 6.4.

SPMD (Single Program, Multiple Data): A programming system that runs a single function on multiple programming elements, but allows each instance of the function to follow different control flow paths. See also ***SIMD, MIMD,*** and ***SIMT***.

stencil pattern: A regular input data access pattern based on a set of fixed offsets relative to an output position. The stencil is repeated for every output position in a grid. This pattern combines the ***map pattern*** with a local ***gather*** over a fixed set of relative offsets and can optionally be implemented using the ***shift pattern***. Stencil operations are common in algorithms that deal with regular grids of data, such as image processing. For example, convolution is an image processing operation where the inputs from a stencil are combined linearly using a weighted sum. See Chapter 7.

step complexity: See ***span complexity***.

strand: In Cilk Plus, a serially executed sequence of instructions that does not contain a spawn or sync point. In the directed acyclic graph model of Section 2.5.2, a strand is a vertex with at most one outgoing and at most one incoming edge. A `cilk_spawn` ends the current strand and starts two new strands, one for the callee and one for the continuation of the caller. A `cilk_sync` ends one or more strands and starts a new strand for the continuation after the join.

strangled scaling: A programming error in which the performance of parallel code is poor due to high contention or overhead, so much so that it may underperform the non-parallel (serial) code. See Section 2.6.4.

strip-mining: When implementing a stencil or map, an optimization that groups instances in a way that avoids unnecessary and redundant memory accesses and aligns memory accesses with vector lanes.

strong scalability: A form of scalability that measures how performance increases when using additional workers but with a fixed problem size. See ***Amdahl's Law*** and ***weak scalability***.

structure-of-arrays (SoA): A data layout for collections of heterogeneous data where all the data for each component of each element of the collection is stored in adjacent physical locations, so that data of the same type is stored together. Compare with ***array-of-structures***.

successor function: In a ***fold***, the function that computes a new state given the old state and a new input item.

superlinear speedup: Speedup where performance grows at a rate greater than the rate at which new workers are added. Since linear scalability is technical optimal, superlinear speedup is typically the result of cache effects, changes in the algorithm behavior, or speculative execution.

superscalar processor: A processor that can execute multiple instructions in a single clock cycle.

superscalar sequence pattern: A sequence of tasks ordered by data dependencies rather than being ordered by a single sequential ordering. This allows parallel (superscalar) execution of tasks that have no relative ordering relative to each other. See Sections 3.6.1.

switch-on-event multithreading: A technique that supports the execution of multiple threads on a single core by switching to another thread on a long-latency event, such as a cache miss.

sync: In terms of Cilk Plus, `cilk_sync` creates a sync point. Control flow pauses at a sync point until completion of all spawns issued by the spawning block that contains the sync point. A sync is not affected by spawns done by other tasks or threads, nor those done prior to entering the current spawning block. An sync is always done when exiting a spawning block that contained any spawns. This is required for program ***composability***.

synchronization: The coordination, of tasks or threads, in order to obtain the desired runtime order. Commonly used to avoid undesired ***race conditions***.

tail recursion: A form of recursion where a result of the recursive call is returned immediately without modification to the parent function. Such uses of recursion can be converted to ***iteration***.

target processor: A (typically specialized) processor to which work can be ***offloaded***. See ***host processor***.

task: A lightweight unit of potential parallelism with its own control flow. Unlike threads, tasks usually do not imply mandatory parallelism. Threads are a mechanism for executing tasks in parallel, whereas tasks are units of work that merely provide the *opportunity* for parallel execution; tasks are not themselves a mechanism of parallel execution.

task parallelism: An attempt to classify parallelism as more oriented around tasks than data. We deliberately avoid use of this term because its meaning varies. In particular, elsewhere "task parallelism" can refer to tasks generated by functional decomposition *or* to irregular tasks still generated by data decomposition. In this book, any parallelism generated by data decomposition, regular or irregular, is considered data parallelism. See Section 2.2.

template metaprogramming: The use of generic programming techniques to manipulate and optimize source code before it is compiled. Specifically, the template rewriting rules in C++ can be interpreted as a functional language for manipulating C++ source code. Some high-performance libraries make use of this fact to automatically perform optimizations of C++ code by, for example, fusing operations together. See the more general term ***metaprogramming***.

temporal locality: Nearby when measured in terms of time; compare with ***spatial locality***. Temporal locality refers to a program behavior in which data is likely to be reused relatively soon. Algorithms exhibiting good temporal locality in data usage can benefit from the data caching common in modern computers. It is not unusual to be able to achieve both temporal and spatial locality in data usage. Computer systems are generally more able to achieve optimal performance when both are achieved, hence the interest in algorithm design to do so.

thread: In general, a *software thread* is any software unit of parallel work with an independent flow of control, and a *hardware thread* is any hardware unit capable of executing a single flow of control (in particular, a hardware unit that maintains a single program counter). Threads are a mechanism for implementing tasks. A multitasking or multithreading operating system will multiplex multiple software threads onto a single hardware thread by interleaving execution via software-created time-slices. A multicore or many-core processor consists of multiple cores to execute at least one independent software thread per core through duplication of hardware. A multithreaded or hyper-threaded processor core will multiplex a single core to execute multiple software threads through interleaving of software threads via hardware mechanisms.

thread parallelism: A mechanism for implementing parallelism in hardware using a separate flow of control for each task. See Section 2.3.

throughput: Given a set of tasks to be performed, the rate at which those tasks are completed. Throughput measures the rate of computation, and it is given in units of tasks per unit time. See ***bandwidth*** and ***latency*** and Section 2.5.1.

tile: A region of memory, typically a section of a larger collection, such as might result from the application of the ***partition pattern***. See ***granularity***, ***block***, and ***tiling***.

tiled decomposition: See ***tiling.***

tiled SIMD: Execution of an ***SPMD*** program using an array of ***SIMD*** processors, each such processor with a separate thread of control.

tiling: Dividing a loop into a set of parallel tasks of a suitable ***granularity***. In general, tiling consists of applying multiple steps on a smaller part of a problem instead of running each step on the whole problem one after the other. The purpose of tiling is to increase the reuse of data in caches. Tiling can lead to dramatic performance increases when a whole problem does not fit in cache. We prefer the term "tiling" to "blocking" and "tile" rather than "block." Tiling and tile have become the more common term in recent times. Sections 5.1.3 and 7.3 for more discussion.

time complexity: A complexity measure for the amount of time used by an algorithm as a function of problem size.

TLB: A Translation Lookaside Buffer is a specialized cache used to hold translations of virtual to physical page addresses. The number of elements in the TLB determines how many pages of memory can be accessed simultaneously with good efficiency. Accessing a page not in the TLB will cause a TLB miss. A TLB miss typically causes a trap to the operating system so that the page table can be referenced and the TLB updated. See Section 2.4.1.

TLB miss: Occurs when a virtual memory access is made for which the page translation is not available in the ***TLB***.

TLB thrashing: The overhead caused by the high ***TLB miss*** rate that results when a program frequently accesses more pages than can be covered by a ***TLB***.

transaction: An atomic update to data, meaning that the results of the update either are not seen or are seen in their entirety. Transactions satisfy the need for atomic data updates to a central repository without requiring an ordering on the updates. Transactions are motivated by the need to have updates be observed in an "all or nothing" fashion. Consider an update to a hotel reservation in an online system, from an "economy room for $75/night" to a "penthouse suite for $9800/night." We do not want a separate task to see a partial update and bill us for $9800/night for an economy room. In general, transaction operations will be non-associative and the outcome will not be deterministic if the order in which the individual operations are performed is non-deterministic. The ***merge***

scatter pattern with a non-associative operator can result in simple forms of the transaction pattern. See Sections 3.7.2 and 6.2.

transactional memory: A way of accessing memory so that a collection of memory updates, called a *transaction*, will be visible to other tasks or threads all at once. Additionally, for a transaction to succeed, any data read during the transaction must not be modified during the transaction by other tasks or threads. Transactions that fail are generally retried until they succeed. Transactional memory offers an alternative method of mutual exclusion from traditional locking that may enhance the scalability of an algorithm in certain cases. Intel Transactional Support Extensions (TSX) support is an example of hardware support for transactional memory.

Translation Lookaside Buffer: See ***TLB***.

uniform parameter: A parameter that is broadcast to all the elements of a map and therefore is the same for each instance of the map's ***elemental function***. See ***varying parameter***.

unpack pattern: The inverse of the ***pack pattern***, this operation scatters data back into its original locations. It may optionally fill in a default value for missing data.

unsplit pattern: The inverse of the ***split pattern***, this operation scatters data back into its original locations. Unlike the case with the ***unpack pattern***, there is no missing data to worry about.

unzip pattern: The inverse of the ***zip pattern***, this operation deinterleaves data and can be used to convert from array-of-structures to structure-of-arrays.

varying parameter: A parameter to a ***map pattern*** that delivers a different element to each instance of the map's ***elemental function***. See ***uniform parameter***.

vector intrinsics: An ***instrinsic*** used to specify a ***vector operation***.

vector operation: A low-level operation that can act on multiple data elements at once in ***SIMD*** fashion.

vector parallelism: A mechanism for implementing parallelism in hardware using the same flow of control on multiple data elements. See Section 2.3.

vector processor: A form of SIMD processor in which large amounts of data are streamed to and from external memory. True vector processors are rare today, so this term now is also used for processors with SIMD instructions that can act on short, fixed-length vectors held in registers.

vectorization: The act of transforming code to enable simultaneous computations using vector hardware. Instructions such as MMX, SSE, and AVX instructions utilize vector hardware. The vectorization of code tends to enhance performance because more data is processed per instruction than would be done otherwise. Vectorization is a specialized form of parallelism. See also ***vectorize***.

vectorize: Converting a program from a scalar implementation to a vectorized implementation to utilize vector hardware such as SIMD instructions (MMX, SSE, AVX, etc.).

vector units: ***functional units*** that can issue multiple operations of the same type in a single clock cycle in ***SIMD*** fashion.

virtual memory: Virtual memory decouples the address used by software from the physical addresses of real memory. The translation from virtual addresses to physical addresses is done in hardware which is initialized and controlled by the operating system. See Section 2.4.1.

VLIW (Very Large Instruction Word): An processor architecture which supports instructions which can explicitly issue multiple operations in a single clock cycle. See ***superscalar processor.***

weak scalability: A form of scalability that measures how performance increases when using additional workers with a problem size that grows at the same rate. See ***Gustafson-Barsis's Law*** and ***strong scalability.***

work: The computational part of a program, as contrasted with communication or coordination. An abstract unit of such computation.

work complexity: The asymptotic number of operations required by an algorithm to run on a single thread. Work complexity is essentially the traditional asymptotic complexity for sequential running time, although frequently, so speedup ratios can be computed, it is better to use ***big Theta notation*** rather than ***big O notation***. Related terms include ***span complexity***.

worker: An abstract unit of actual parallelism, for example, a ***core*** or a ***SIMD lane***.

working set: For an algorithm, the set of data that should be maintained in cache for good performance.

work-span: A model for parallel computation that can be used to compute both upper and lower bounds on speedup. See Section 2.5.6. Related terms include ***span complexity*** and ***work complexity***.

workpile pattern: An extension of the ***map pattern*** that allows new work items to be added during execution from inside the elemental function. If the map pattern can be thought of as a ***parallelization*** of a `for` loop, the workpile pattern can be thought of as a generalization of a `while` loop. See Section 3.6.4.

work-stealing: A ***load balancing*** technique where /workers/ that become idle search for and “steal” pending work from other, busy workers.

zip pattern: A special case of the ***gather pattern*** that interleaves elements from collections, converting from structure-of-arrays to array-of-structures. See Section 6.1.3.

Computers / Parallel Programming

Structured Parallel Programming

Patterns for Efficient Computation

Michael McCool
Arch D. Robison
James Reinders

A Developer's Guide to Patterns for High-Performance Parallel Programming

"...a groundbreaker, it presents real-life algorithms and the issues and solutions to get them to profit from the coming multi/many core evolution."

—**MICHELE DELSOL,** Parallel Programming Consultant and Instructor

Programming is now parallel programming. Multicore processors are now standard, and all developers need to learn the fundamentals of parallel algorithm design. However, much as structured programming revolutionized traditional serial programming decades ago, a new kind of structured programming, based on patterns, is relevant to parallel programming today.

This book explains how to design and implement maintainable and efficient parallel programs using a pattern-based approach. It presents both theory and practice, drawing on multiple programming models in detailed, concrete examples that will help you learn and apply efficient patterns in your applications.

Most of the many included examples use two of the most popular and cutting-edge programming models for parallel programming in C++: Threading Building Blocks and Cilk Plus. These portable programming models enable easy integration into existing applications, preserve investments in existing code, and speed the development of parallel applications. In short, this book:

- Offers structure and insight that you can apply to a variety of parallel programming models
- Develops a composable, structured, scalable, and machine-independent approach to parallel computing
- Includes detailed examples in both Cilk Plus and the latest Threading Building Blocks, which support a wide variety of computers

Michael McCool
Software Architect, Intel Corporation and Adjunct Associate Professor, University of Waterloo

Arch D. Robison
Architect of Threading Building Blocks, Intel Corporation

James Reinders
Senior Engineer, Intel Corporation

ISBN: 978-0-12-415993-8
90000
9 780124 159938

MORGAN KAUFMANN PUBLISHERS
AN IMPRINT OF ELSEVIER
mkp.com