CHAPTER

1 Introduction

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

Structured Parallel Programming. DOI: 10.1016/B978-0-12-415993-8.00001-3

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).

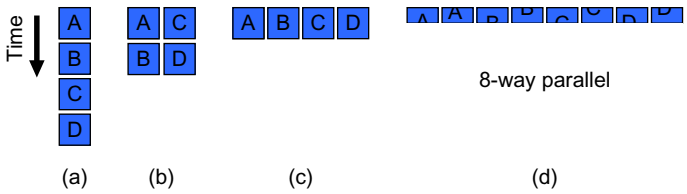

FIGURE 1.1

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.

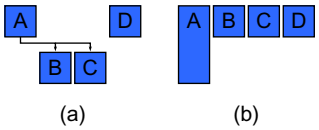

FIGURE 1.2

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, the sum of the running time of tasks A and C, or the running time of D, 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× every two years, or 1.995 × every 2 years. The data shows that Moore's prediction of 2x every two years has been amazingly accurate. While we only give data for Intel processors, processors from other vendors have shown similar trends.

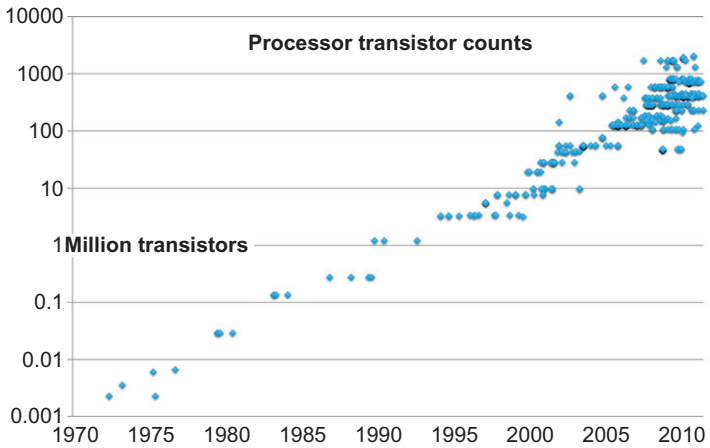

FIGURE 1.3

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.

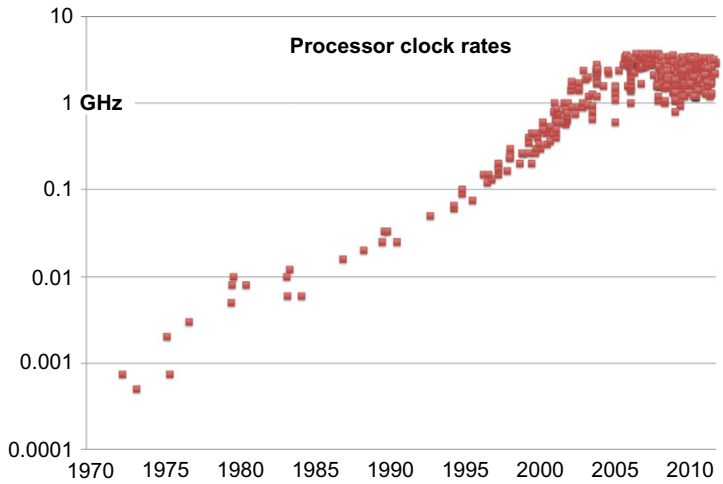

FIGURE 1.4

Growth of processor clock rates over time (log scale). This graph shows a dramatic halt by 2005 due to the power wall, although current processors are available over a diverse range of clock frequencies.

The power wall results because power consumption (and heat generation) increases non-linearly as the clock rate increases. Increasing clock rates any further will exceed the power density that can be dealt with by air cooling, and also results in power-inefficient computation.

The second wall is the instruction-level parallelism (ILP) wall. Many programmers would like parallelization to somehow be done automatically. The fact is that automatic parallelization *is already* being done at the instruction level, and has been done for decades, but has reached its limits. Hardware is naturally parallel, and modern processors typically include a large amount of circuitry to extract available parallelism from serial instruction streams. For example, if two nearby instructions do not depend on each other, modern processors can often start them both at the same time, a technique called **superscalar instruction issue**. Some processors can issue up to six instructions at the same time (an example being the POWER2 architecture), but this is about the useful limit for most programs on real processors. Analysis of large code bases show that on average there is not much more available superscalar parallelism at the instruction level than this [BYP+91, JW89, RDN93, TEL95]. More specifically, more parallelism may be available, but it is bursty or otherwise hard to use in a sustained way by real processors with finite resources. A related technique is **Very Large Instruction Word** (VLIW) processing, in which the analysis of which instructions to execute in parallel is done in advance by the compiler. However, even with the help of offline program analysis, it is difficult to find significant sustained parallelism in most programs [HF99] without diminishing returns on hardware investments. Modern processors also use **pipelining**, in which many operations are broken into a sequence of stages so that many instructions can be processed at once in an assembly-line fashion, which can greatly increase the overall instruction processing throughput of a processor. However, pipelining is accomplished by reducing the amount of logic per stage to reduce the time between clocked circuits, and there is a practical limit to the number of stages into which instruction processing

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 once 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

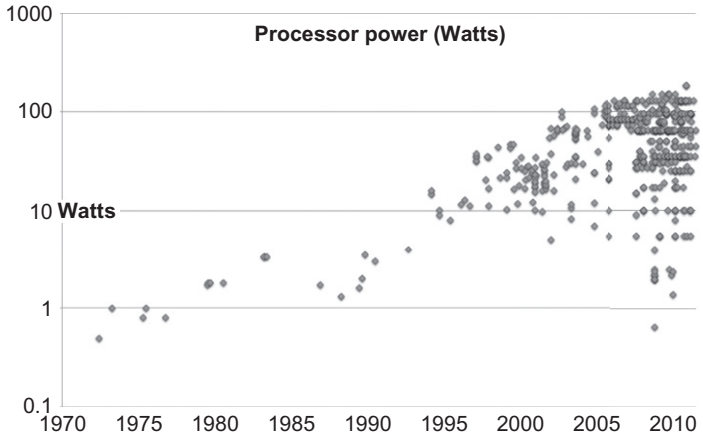

FIGURE 1.5

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.

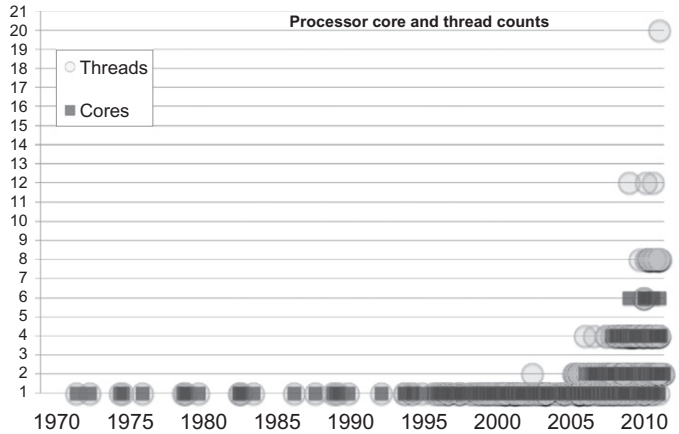

FIGURE 1.6

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

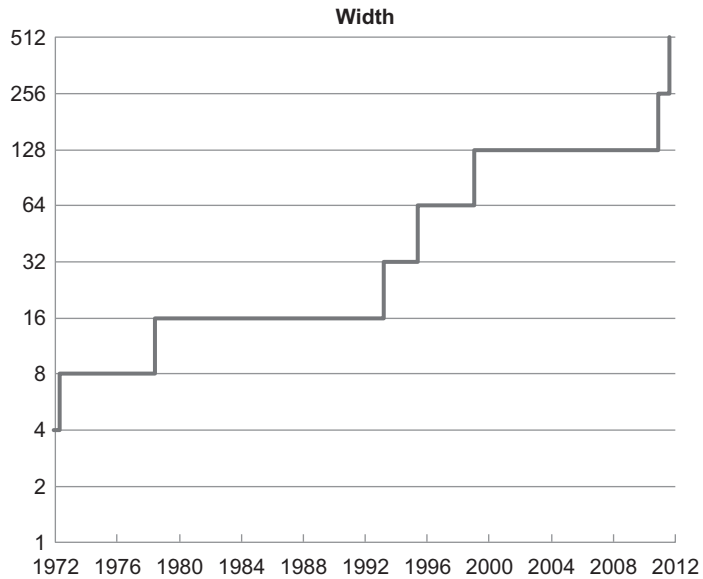

FIGURE 1.7

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.

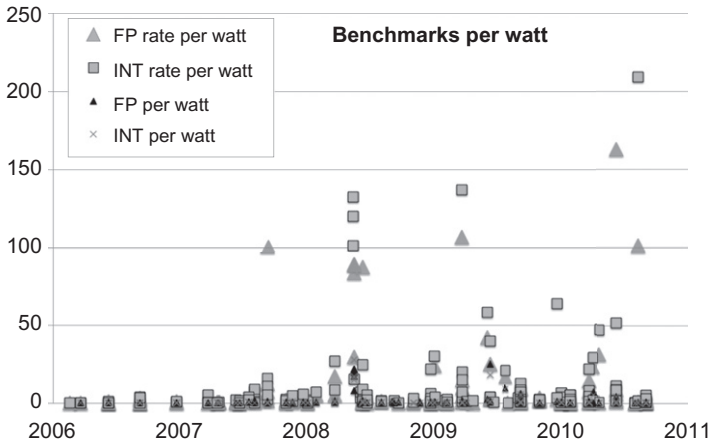

FIGURE 1.8

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.

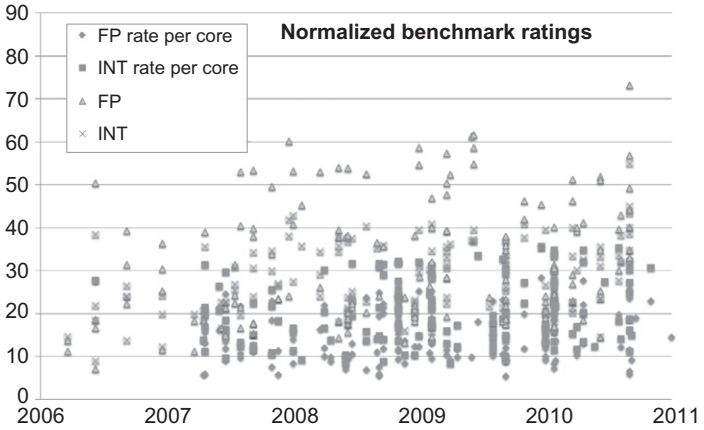

FIGURE 1.9

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.

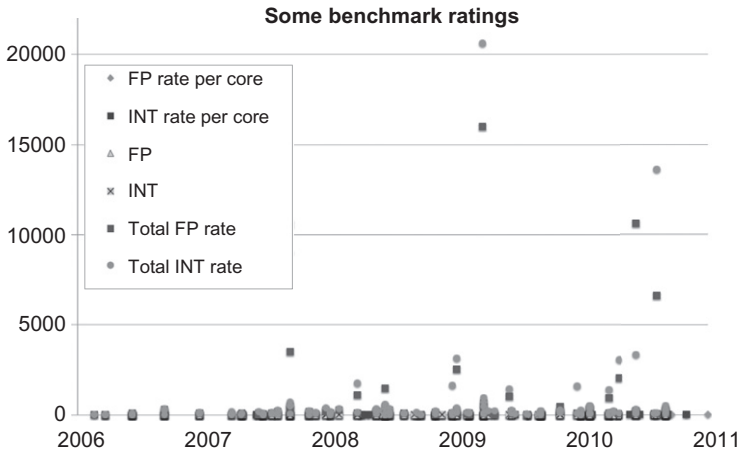

FIGURE 1.10

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

```
1  void
2  addme(int n, double a[n], double b[n], double c[n]) {
3     int i;
4     for (i = 0; i < n; ++i)
5        a[i] = b[i] + c[i];
6  }
```

LISTING 1.1

Add two vectors in C, with implied serial ordering.

```
1  double a[10];
2  a[0] = 1;
3  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.

```
1  void
2  addme(int n, double a[n], double b[n], double c[n]) {
3     a[:] = b[:] + c[:];
4  }
```

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

```
1  double summe(int n, double a[n]) {
2     double mysum = 0;
3     int i;
4     for (i = 0; i < n; ++i)
5        mysum += a[i];
6     return mysum;
7  }
```

LISTING 1.4

An ordered sum creates a dependency in C.

```
1  double summe(int n, double a[n]) {
2     sum_reducer<double> mysum (0);
3     cilk_for (int i = 0; i < n; ++i)
4        mysum += a[i];
5    return mysum.get_value();
6  }
```

LISTING 1.5

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

```
1  void callme() {
2     foo();
3     bar();
4  }
```

LISTING 1.6

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

says that reordering 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.

```
1 void callme() {
2    cilk_spawn foo();
3    bar();
4 }
```

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.

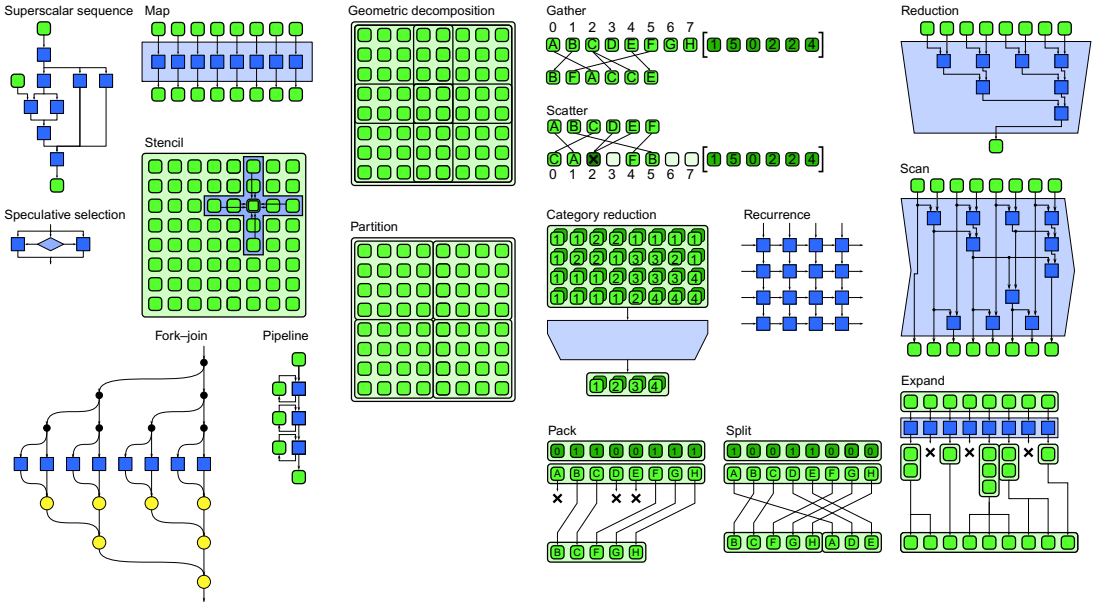

FIGURE 1.11

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.

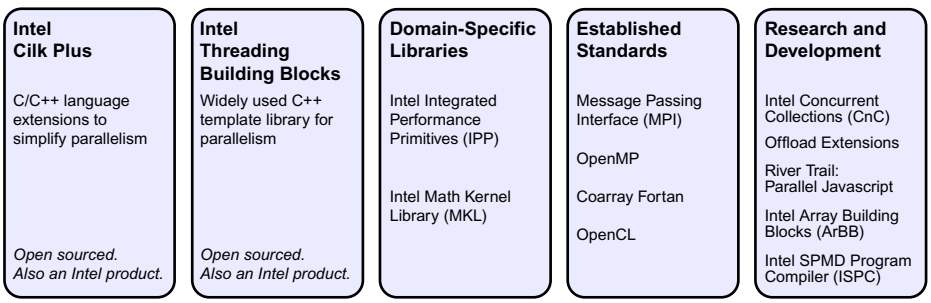

FIGURE 1.12

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 is 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

```
1  for (i = 0; i < 10000; ++i) {
2      a[i] = b[i] + c[i];
3  }
```

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 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-

```
1  float my_simple_add(float x1, float x2) {
2     return x1 + x2;
3  }
4  ...
5  for (int j = 0; j < N; ++j) {
6    outputx[j] = my_simple_add(inputa[j], inputb[j]);
7  }
```

LISTING 1.11

Scalar function for addition in C.

```
1  __declspec(vector)
2  float my_simple_add(float x1, float x2) {
3     return x1 + x2;
4  }
5  ...
6  #pragma simd
7  for (int j = 0; j < N; ++j) {
8    outputx[j] = my_simple_add(inputa[j], inputb[j]);
9  }
```

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 have to 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 have to 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.

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

LISTING 1.13

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

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

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.

```
1  void arbb_vector (
2     dense<f32>& A,
3     dense<f32> B,
4     dense<f32> C,
5     dense<f32> D
6  ) {
7     A += B - C/D;
8  }
9
10 dense<f32> A, B, C, D;
11 // fill A, B, C, D with data ...
12
13 // invoke function over entire collections
14 call(arbb_vector)(A,B,C,D);
```

LISTING 1.15

Vector computation in ArBB.

```
1  void arbb_map (
2     f32& a, // input and output
3     f32 b,  // input
4     f32 c,  // input
5     f32 d   // input
6  ) {
7     a += b - c/d;
8  }
9
10 void arbb_call (
11    dense<f32>& A, // input and output
12    dense<f32> B,  // input
13    dense<f32> C,  // input
14    f32 d          // input (uniform; will be replicated)
15 ) {
16    map(arbb_map)(A,B,C,d);
17 }
```

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 a 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.