Foundations

We begin our study of parallel programming by building a solid foundation. The most important goal is to clarify the difference between the sequential and parallel programming worlds. In sequential computing, operations are performed one at a time, making it straightforward to reason about the correctness and performance characteristics of a program. In parallel computing many operations take place at once, complicating our reasoning about correctness and performance, and as a result, modifying our programming approach. This part explains the main consequences of this distinction.

Our introduction to parallel computation in Chapter 1 begins by solving a simple problem of counting the number of occurrences of 3 in a 1-dimensional array. This trivial task requires four attempts before we create a program with reasonable performance. Even then, we find that our maximum hoped-for speedup can't be realized. While working through the example, we introduce a series of basic concepts of parallelism.

Chapter 2 describes the basic architectural features of parallel computers. It is an interesting topic in its own right, because challenging problems such as interprocessor communication have a multitude of potential solutions, and the techniques that architects use to address them exhibit considerable ingenuity. The main conclusion of our tour of parallel machines will be that they are extremely different. Because programmers need to know certain properties of the underlying machine to write quality programs, it will be necessary to find a machine model that unifies the disparate architectures. We introduce such a model as the basis for our subsequent study.

With a clear idea of how parallel computers work, Chapter 3 characterizes the many conceptual issues surrounding parallel performance. We introduce key ideas including latency, bandwidth, speedup and efficiency. Certain facets of programming, such as dependences, are highlighted as being a source of interference among parallel threads. Once these foundational ideas have been introduced, we will be prepared to move on to the algorithmic ideas presented in Part 2.
Parallel computation is a fundamental technique by which computations can be accelerated, so the increasing availability of parallel hardware represents a tremendous opportunity. But implementing a parallel solution presents certain conceptual and programming challenges that this textbook is designed to address. To place the opportunities and challenges in perspective, this chapter sets the context and introduces basic ideas.

The Power and Potential of Parallelism

Parallelism arises frequently in everyday life. More importantly, parallelism has contributed in many ways to the steady performance improvement in computers over the past several decades. And now, new opportunities are available. Let's look closely.

Parallelism, a Familiar Concept

Parallelism is a familiar concept. Juggling is a parallel task that humans can perform. House construction is a parallel activity, because several workers can perform separate tasks simultaneously, such as wiring, plumbing, and furnace duct installation, and so on. Most manufacturing—cars, hairdryers, frozen dinners—is performed in parallel using an assembly line, or pipeline, in which many units of the product are under construction at once. A call center, where many employees service customers at the same time, is another organization that applies parallelism.

Although familiar, these forms of parallelism are different. The call center, for example, differs from house construction in a fundamental way: Calls are generally independent and can be serviced in any order with little interaction among the workers. In construction, some tasks can be performed simultaneously—wiring and plumbing—while others are ordered—framing must precede wiring. The ordering restricts the amount of parallelism that can be applied at once, limiting the speed at which a construction project can complete. The ordering also increases the degree
of interaction among the workers. Manufacturing pipelines are different still, because they generally have strict ordering constraints with the separate stages often being performed sequentially; the parallelism comes from having many instances of the product in the pipeline at once. And juggling is an instance of event-driven parallelism, where an event—a falling ball—causes the execution of operations—catching, throwing—in response to the event. Such familiar forms of parallelism will also arise in our consideration of parallel computation.

Parallelism in Computer Programs

The main motivation for executing program instructions in parallel is to complete a computation faster. But most programs today are incapable of much improvement through parallelism, because they were written assuming that the instructions would be executed in order, one at a time, that is, sequentially. The semantics of most programming languages embed sequential execution, and the resulting programs typically rely so heavily on this property for their correctness that it is rare to find significant opportunities for parallel execution. To be sure, there are some opportunities, as when the expression \((a+b) \times (c+d)\) must be evaluated; assuming these are simple variables, the subexpressions \((a+b)\) and \((c+d)\) are independent of each other, so they can be computed simultaneously. Such opportunities are an example of Instruction Level Parallelism (ILP).

Indeed, one reason that we have continued to write sequential programs is because computer architects have been so successful at exploiting parallelism. They have used the steady improvements in silicon technology to add several kinds of parallelism, including ILP, into sequential processor design. First, architects provide separate wires and caches for instructions and data. The separation allows instruction and data memory references to execute in parallel without interfering. Second, instruction execution is pipelined, fetching and decoding future instructions while the current instruction is being executed and while the results of past instructions are still being written to memory. Furthermore, the processors issue (initiate) more than one instruction at a time, they prefetch instructions and data, they speculatively perform operations in parallel even if they cannot be sure that they will be needed, and they use highly parallel circuits to perform basic arithmetic operations. In short, modern processors are highly parallel systems.

The key point for programmers is that all of this parallelism has been transparently available to sequential programs. We call this hidden parallelism. Such parallelism, together with increasing clock speeds, has allowed each succeeding generation of processor chip to execute programs faster, while preserving the illusion of sequential execution. But the prospects for finding new opportunities to apply parallelism while preserving sequential semantics are becoming limited. More seriously, existing techniques for exploiting ILP have largely reached the point of diminishing returns, in terms of both power consumption and performance. So, given current
technologies, sequential program execution may be approaching its maximum speed.

To continue achieving significant performance improvements, we must move beyond the single sequence of instructions typical of existing programs. We need programs that have multiple instruction streams that operate simultaneously. This approach will require new programming techniques, which is the topic of this book.

**Multi-Core Computers, an Opportunity**

Though performance improvements for a single processor may be reaching a limit, the prophecy of Moore’s Law continues to deliver improved transistor densities. Chip manufacturers have used this opportunity to place more than one instruction execution engine, together with its caches, on a single chip. This structure has rapidly acquired the name *core*, because it represents the core components of a typical sequential processor. Early chips had 2, 4, or 8 cores, but this number increases with each generation.

The advent of the first multi-core chips in 2005/2006 prompted a community-wide discussion about the “end of the free lunch.” The key observations of the discussion were as follows:

- Software developers have enjoyed steadily improving performance for decades—the “free lunch”—thanks to advances in silicon technology and architecture design (hidden parallelism), as just described.
- Programmers, not needing to be concerned with performance, have changed their techniques and methodologies little over the years (the object-oriented paradigm is a notable exception).
- Existing software generally cannot exploit multi-core chips directly.
- Programs that cannot exploit multi-core chips do not realize any performance improvements now and they will not in the future.
- Most programmers do not now know how to write parallel programs.

The uncomfortable conclusion was that programs need to change, and to make that happen, programmers do, too.

Though the conclusion might be viewed by some as bad news, there was corresponding good news. Specifically, if a computation is rewritten to be parallel and if the parallel program is also *scalable*, meaning that it is capable of using progressively more processors, then as silicon technology advances and more cores are added to future chips, the rewritten program will stay on the performance curve. Non-scalable parallel programs, though, will not enjoy the continued benefits of silicon technology advances. It is important, therefore, to achieve scalable parallelism.

Some observers, especially in the graphics community, no doubt find the discussion of the need for parallel computation curious, because they have been using parallelism for years. Graphics processing units (GPUs), a.k.a. graphics cards, have been
the standard technique for accelerating the rendering pipeline. Though the GPU might seem like a niche co-processor of little interest for general computer applications, advances in silicon technology have enabled the notion of GPGPU, general-purpose computing on graphics processing units. With a generation cycle of roughly 18 months, the GPU has steadily become increasingly general with each generation, and parallel programmers have applied them to a long list of compute-intensive non-graphics applications. Like multi-core chips, exploiting the potential of GPUs requires knowledge of parallel programming.

Even More Opportunities to Use Parallel Hardware

The opportunities to use parallelism discussed so far involve a small number of processors. But there are many opportunities that are more ambitious.

Supercomputers. The problems of interest at the national research labs, the military, and large corporations have traditionally required supercomputers, which by definition are the world’s fastest computers. Twenty years ago, supercomputers were custom-made single-processor systems (generally with vector processing capabilities), but single-processor systems last appeared on the Top 500 List of fastest computers in November 1996, when just three appeared, ranking #265, #374, and #498. Today, the Top 500 List is dominated by parallel computers with many thousands of processors. In many ways, supercomputer programmers form the largest, most experienced community of parallel programmers.

Clusters. It is often observed that no matter how fast a single computer is, connecting two or more of them together produces a faster computer in the sense that the combined machine can execute more instructions per unit time. Of course, well-written parallel programs are needed to exploit the added power. Clusters have been popular since the 1990s because they are relatively inexpensive to build from commodity parts. The low price not only makes them attractive for small groups—labs or small firms—but also gives them a tremendous price/performance advantage over other forms of high-end computing. In the June, 2007 Top 500 List (www.top500.org), clusters represented 74.6 percent of the list.

Servers. The expansion of the Internet and the popularity of remote services, such as searching, have created huge installations of networked computers. In terms of total number of instructions executed per second, these centers represent a huge computational resource. The typical computations—the processing of search queries, for example—are independent of each other; furthermore they use distributed—as opposed to parallel—programming techniques (see the next section, Parallel Computing versus Distributed Computing). Nevertheless, these huge networked systems are being used to analyze the features of their workload and to perform other data-intensive computations; the solutions also apply parallel programming techniques.
Grid Computing. Generalizing still further, the collection of computers need not be in the same location, nor administered by the same organization; after all, the computers connected by the Internet represent an enormous computing resource. By analogy with the power grid, a computing grid seeks to provide a single convenient computing service, even though the underlying computer typically consists of physically dispersed machines governed by multiple administrative organizations. Many technical issues remain before grids become commonplace, but they are a topic of active research.

We see then, that there are ample opportunities to use a parallel program beyond the few processors on a single silicon chip. These large computer systems also motivate us to write scalable parallel programs.

Parallel Computing versus Distributed Computing

As suggested above, distributed computing and parallel computing are different.

The goal of parallel computing has traditionally been to provide performance—either in terms of processor power or memory—that a single processor cannot provide; thus, the goal is to use multiple processors to solve a single problem. The goal of distributed computing is to provide convenience, where convenience includes availability, reliability, and physical distribution (being able to access the distributed system from many different locations).

In parallel computation the interaction among processors is generally frequent, typically fine grained with low overhead, and assumed to be reliable. In distributed computation the interaction is generally infrequent, heavier weight, and assumed to be unreliable. Parallel computation values short execution time; distributed computation values long uptime.

Of course, parallel computing and distributed computing are closely related. Some features are a matter of degree—frequency of interaction among processors—and we haven’t specified the crossover point. Some features are a matter of emphasis—speed versus reliability—and we know that both properties are important to both types of systems. It follows then, that the two kinds of computing represent distinct, but nearby points in a multidimensional space. The more one knows about parallel computation (or distributed computation, but that’s not the emphasis of this book), the more easily one can move around in the parallel-distributed space. Learning the basics of parallel computation will be valuable even for programmers with no need to improve performance.

System Level Parallelism

Return for a moment to our earlier argument that to enjoy the benefits from parallelism we must move beyond a single sequence of instructions. This argument is relevant within a single application. When we view a desktop computer’s software from a system level, however, we see many tasks executing at once. The operating
system orchestrates their concurrent execution, which heretofore meant that several tasks were juggled at once, with only one executing at a time, a technique known as multitasking. An obvious question is, "Why not simply run these separate tasks on the extra processors?" It makes sense because their concurrent design ensures that whatever interactions are required among them will be handled safely.

**Concurrency and Parallelism.** Though these terms are closely related, history influences how we use them. Concurrency is widely used in the operating systems and database communities to describe executions that are logically simultaneous, while parallelism is typically used by the architecture and supercomputing communities to describe executions that physically execute simultaneously. In either case, the codes that execute simultaneously exhibit unknown timing characteristics. For example, an operating system might execute just one code segment at a time, but because the execution can context switch at any point in time to another code segment, the issues are the same as if they were physically executing simultaneously. Similarly, with physical parallelism, the timing relationships among the code segments is unpredictable, forcing the assumption that any timing is possible. Thus, the issues for program correctness are the same for both concurrent and parallel computation, and in a sense, they are endpoints on a spectrum of varying parallel resources. In this book, we will use the terms interchangeably to refer to logical concurrency.

The first answer is that for the large-scale parallelism just described, there are not nearly enough tasks to keep large processors busy. But, for small amounts of parallelism as is typical of today’s multi-core chips, the separate tasks can be run on separate processors. Indeed, it has been suggested that continuously executing tasks such as security software (or the OS itself!) would be good candidates for the extra processors. But there may be fewer opportunities than might at first appear. First, many applications don’t stress the hardware even now—word processors perform spell-checking continuously in the background and never fall behind the typist. Second, much of the multitasking in an operating system comes from switching to a new task when the currently executing task requests a time-consuming, external operation, such as a page fault, disk I/O, or network I/O. Suppose that task A blocks on such a request. On a single processor system, some task B could execute while A blocks, providing good utilization of the processor, but on a multi-processor system, task B may have already completed execution on a separate processor, forcing the processor that executes task A to simply remain idle.

The main reason that running multiple tasks on the separate parallel processors is not a silver bullet, however, is because it doesn’t usually improve the performance of an individual application. And in those cases where improved performance is needed, it is essential that there are multiple streams of cooperating instructions that use multiple processors effectively.

---

1Because certain I/O devices like disk controllers are typically separate from the main execution engine, there has long been true parallel execution among the processor and its external devices.
Convenience of Parallel Abstractions

Finally, there is one reason to exploit parallelism besides performance: Some computations are more easily expressed as parallel computations. For example, user interfaces are typically best written as a collection of threads, with one thread responsible for interacting with the user: The thread waits for user input and dispatches other threads to respond appropriately. With such an organization, the code that displays a widget is greatly simplified because, for example, it needn’t concern itself with the responsibility of polling for a user mouse click that might come at any time.

As we shall see, the abstractions used to organize and manage parallel computations make it convenient and safe to use multiple instruction streams. When flow of control is unpredictable, parallelism can help even when the resulting instruction sequences are not executed simultaneously. Thus, while we emphasize the fast solution of a problem, we acknowledge that there are other uses of parallelism.

Examining Sequential and Parallel Programs

The previous sections have emphasized the potential advantages available in hardware. We have asserted that existing sequential programs cannot take advantage of, say, multi-core computers, so it’s time to consider ways to realize the benefits of parallel hardware.

Parallelizing Compilers

Knowing that a compiler translates the programs that we write into the machine instructions of the computer that we use, and not knowing (at least for most of us) how this magical translation is done, it is reasonable to wonder why someone doesn’t just write a compiler that translates existing programs into a form suitable for parallel execution. After all, the sequential program specifies the computation, and all that needs to be done is to transform the same operations into a parallel form. This idea to compile sequential programs for parallel machines was among the first approaches tried, and it continues to be a dream. Unfortunately, the dream seems beyond reach, despite over three decades of intense research.

The reason for pessimism is that scalable parallel algorithms are generally qualitatively different from the sequential algorithms found in existing programs. We will describe this situation by saying that the parallel solution typically requires a paradigm shift in the solution approach. Since compilers transform programs in ways that preserve their correctness, they do not change the essential features of the algorithm. (Figure 1.1 illustrates the phases of a generic compiler.) Compilers change the form of the program code; they can remove unnecessary instructions, as for example, when 0 is added to a variable; they can add helpful instructions, say, to check that array indices are in bounds; they can move instructions around, say hoisting them out of loops when the value computed isn’t affected by the iteration;
and they can perform other amazing transformations. But the general algorithm is preserved. Whether it was sequential or parallel in the source form, the algorithm will remain fundamentally the same in the object form.

Thus, although automatic parallelization by compiler would be wonderful, we must consider other approaches. First, consider how sequential and parallel algorithms for the same task might differ.

**A Paradigm Shift**

To make it clear that sequential and parallel algorithms are different, compare alternative algorithms for finding the sum of a sequence of numbers. This example is sufficiently simple that there are compiler techniques to identify it and generate a more parallel solution, but we choose it because it is a simple illustration of the conceptual difference between a sequential solution and a parallel solution.

To begin, we assume that the sequence has \( n \) data values, \( x_0, x_1, x_2, \ldots, x_{n-1} \) and that these have been stored in an array, \( x \).

**Iterative Sum.** Perhaps the most intuitive solution is to initialize a variable, call it \( \text{sum} \), to 0 and then iteratively add the elements of the sequence. Such a computation is typically programmed using a loop with an index value to reference the elements of the sequence, as follows:

\[
\begin{align*}
1 & \quad \text{sum} = 0; \\
2 & \quad \text{for}(i=0; i<n; i++) \\
3 & \quad \{
4 & \quad \text{sum}+=x[i]; \\
5 & \quad \}
\end{align*}
\]

This computation can be abstracted as a graph showing the order in which the numbers are combined (see Figure 1.2). Such solutions are intuitive because most of us first learn programming in this style.

Of course, addition over the real numbers is an associative and commutative operation, implying that its values need not be summed in the order specified, least index to greatest index. We can add them in another order—perhaps one that admits more parallelism—and get the same answer.

**Nonassociativity.** Strictly speaking, addition is not associative on the fixed precision representation of a floating point numbers, because they only approximate real numbers. For some sequences of values, different orders of addition will produce different answers. We ignore such issues and reorder computations to improve performance, reasoning that a) under most circumstances the sequence's order was arbitrary in the first place, and, b) in those cases where it is not arbitrary and numerical precision is a potential issue, error management is required throughout the computation.
Figure 1.1
Generic compilation process. In the first phases, the source program is scanned (lexical analysis) and parsed (syntactic analysis), resulting in a program representation known as an Abstract Syntax Tree. In this form the program is type-checked to ensure, for example, that variables are declared. Next, the program is transformed into a linear sequence of simple instructions known as 3-address code. The resulting intermediate representation is improved (grandly called optimization). The resulting code is transformed into machine-specific assembly code. It is then trivial to transform the result into binary code and to assign virtual addresses.
Examining Sequential and Parallel Programs

Type Check \(\Rightarrow\) Decorated AST \(\Rightarrow\) Linearize as 3-Address Code \(\Rightarrow\) Intermediate representation

... ifStmt
  expr assign
    eqTest lhs expr
      a 0
      (int,local)(int,literal)
  (int,local)
  div
    x
    2
      x
      (int,local)

Assembly code \(\Rightarrow\) Assemble and link \(\Rightarrow\) Binary

ld 8,a_offset(fp)
bnez 8,L1
ld 9,x_offset(fp)
sra 9,1
st 9,x_offset(fp)
L1:
...

... 00110000
     11101001
     10110100
     01000000
     00111010
...
Pair-Wise Summation. Another, more parallel order of summation is to add even/odd pairs of data values yielding the intermediate sums,

$$(x_0 + x_1), (x_2 + x_3), (x_4 + x_5), (x_6 + x_7), \ldots$$

which are added in pairs,

$$((x_0 + x_1) + (x_2 + x_3)), ((x_4 + x_5) + (x_6 + x_7)), \ldots$$

yielding more intermediate sums, which are themselves added in pairs, and so on. This solution can be visualized as inducing a tree on the computation, where the original data values are leaves, the intermediate nodes are the sum of the nodes below them, and the root is the overall sum (see Figure 1.3).

Comparing Figures 1.2 and 1.3, we see that because the two solutions require the same number of operations and the same number of intermediate sums, there is no
time advantage to either solution when using one processor. However, with a parallel computer that has at least \( P = n/2 \) processors, all of the additions at the same level of the tree can be computed simultaneously, yielding a solution with time complexity that is proportional to \( \log n \). The strategy is a significant improvement over the linear time sequential algorithm. Like the sequential solution, the pair-wise approach is a very intuitive way to think about the computation.

**Expressing Parallel Sum.** The iterative summation was illustrated using C code, but the pair-wise summation was not. If we are not concerned about writing code for an arbitrary length array, we might write it as follows to highlight the binary tree structure of the computation:

```c
1  t[0]=x[0]+x[1];
5  t[4]=t[0]+t[1];
7  sum=t[4]+t[5];
```

The first four assignments can be performed in parallel; after they are complete, the next two (5, 6) can also be performed in parallel.

**Parallel Prefix Sum**

Closely related to the sum is the prefix sum, also known as scan in many parallel programming languages. It begins with the same sequence of \( n \) values,

\[ x_0, x_1, x_2, \ldots, x_{n-1} \]

but the desired computation is the sequence

\[ y_0, y_1, y_2, \ldots, y_{n-1} \]

such that each \( y_i \) is the sum of the first \( i \) elements of the input, that is,

\[ y_i = \sum_{j \leq i} x_j \]

Solving the prefix sum in parallel is less obvious than summation, because all of the intermediate values of the sequential solution are needed. It seems as though there is no advantage to, nor much possibility of, finding better solutions. But in fact the prefix sum can be performed in parallel.

The observation is that the summation by pairs approach can be modified to compute the prefix values. The idea is that each leaf processor storing \( x_i \) could compute the value, \( y_i \), if it only knew the sum of all elements to its left, that is, its prefix; in the course of summing by pairs, we know the sum of all subtrees (see Figure 1.3), and if we save that information, we can determine the prefixes without directly summing them. To do so, we start at the root, whose prefix—that is, the sum of all elements before the elements of the sequence—is 0. This is also the prefix of its left subtree, and the total for its left subtree is the prefix for the right subtree. Applying this idea
inductively, we get the following set of rules:

- Compute the grand total at the root by pair-wise sum, as before.
- On completion, imagine the root receiving a 0 from its (nonexistent) parent.
- All non-leaf nodes receive a value from their parent, relay that value to their left child, and send their right child the sum of the parent’s value and their left child’s value that was computed on the way up; these are the prefixes of their child nodes.
- Leaves add the prefix value from above and the saved input.

The values moving down the tree are the prefixes for the child nodes (see Figure 1.4, where downward moving prefix values are shown in the white square).

The computation is known as the parallel prefix computation. It requires an up sweep and a down sweep in the tree, but all operations at each level in a sweep can be performed concurrently. At most two add operations are required at each node, one going up and one coming down, plus the routing logic. Thus, the parallel prefix also has logarithmic time complexity. Many seemingly sequential operations yield to the parallel prefix approach.

An essential difference between the sequential and parallel algorithms is that we organized the parallel algorithms to change the order of the computation.

**Figure 1.4**
Computing the prefix sum. The gray node values, computed going up the tree, are from the pair-wise sum algorithm; the white values, the prefixes, are computed going down the tree by a simple rule: send the value from the parent to the left child; add the sum from the left child (that came up) to the value from the parent and send it to the right child.
Parallelism Using Multiple Instruction Streams

In this section, we illustrate the complexities of parallel programming by developing a parallel program that solves a trivial problem. It will take us four tries to get a satisfactory result.

We begin by describing one way to conceptualize an instruction stream.

The Concept of a Thread

A thread, or thread of execution, is a unit of parallelism. As we will discuss in Chapter 3, a thread has everything needed to execute a stream of instructions—a private program text, a call stack, and a program counter—but it shares access to memory with other threads. Thus, multiple threads can cooperate to compute on global data.

For example, the iterative summation loop previously discussed could be the basis for a thread if we rewrote it as follows:

```c
1  for(i=start; i<end; i++)
2  {
3    sum+=x[i];
4  }
```

The loop index \( i \) would be local to the call stack, while the variable \( \text{sum} \) and the array \( x \) would be shared. By assigning each thread a different set of \( \text{start} \) and \( \text{end} \) values, multiple threads can work on the problem at once, introducing a form of parallelism.

A Multithreaded Solution to Counting 3s

To understand the obstacles to writing correct, efficient and scalable threaded programs, consider the problem of counting the number of 3s in an array. This computation can be trivially expressed in most sequential programming languages; what is required to solve it using threads?

The Parallel Computer. To make matters concrete, let’s assume that we will execute our parallel program on a parallel computer with eight processors, as shown in Figure 1.5. Consider the two processors labeled P0 and P1. Each is shown adjacent to its private Level 1 cache, labeled L1. A cache is fast (compared to the RAM) memory for storing instructions and data while a program runs. Each pair of processors (P0 and P1, P2 and P3, and so on) shares a Level 2 cache, which is even larger but slower than the L1 caches. Finally, all eight processors share the Level 3 cache, which is larger and slower than the L2 cache but still faster than RAM. Data can be shared among the processors by exchanging information in the L2 and L3 caches.
First Solution: Try 1. We will use a threads programming model in which each thread executes on a dedicated processor, and the threads communicate with one another through shared memory (including the caches). Thus, each thread has its own process state, but all threads share memory and file state. The serial code to count the number of 3s follows:

```c
1 int *array;
2 int length;
3 int count;
4
5 int count3s()
6 {
7   int i;
8   count=0;
9   for(i=0; i<length; i++)
10   {
11     if(array[i]==3)
12     {
13       count++;
14     }
15   }
16   return count;
17 }
```
To implement a parallel version of this code, we can partition the array so that each thread is responsible for counting the number of 3s in $1/t$ of the array, where $t$ is the number of threads. Figure 1.6 shows graphically how we might divide the work for $t=4$ threads and $length=16$.

We can implement this logic with the function `thread_create()`, which takes two arguments—the name of a function to execute and an integer that identifies the thread’s ID—and spawns a thread that executes the specified function with the thread ID as a parameter. The resulting program is shown in Figure 1.7.

```c
1  int t;                /* number of threads */
2  int *array;
3  int length;
4  int count;
5  
6  void count3s()
7  {
8    int i;
9    count = 0;
10   /* Create t threads */
11   for(i=0; i<t; i++)
12     {
13      thread_create(count3s_thread, i);
14     }
15   return count;
16  }
17
18  void count3s_thread(int id)
19  {
20     /* Compute portion of the array that this thread
21        should work on */
22     int length_per_thread=length/t;
23     int start=id*length_per_thread;
24     for(i=start; i<start+length_per_thread; i++)
25     {
26       
(continued)
```
Chapter 1  Introduction

Unfortunately, this seemingly straightforward code will not produce the correct answer because there is a *race condition* in the statement that increments the value of `count` on line 29. A race condition exists when the result of an execution depends on the timing of two or more events. In this case, the problem arises because the statement that increments `count` is typically implemented on modern machines as a series of primitive machine instructions:

- Load `count` into a register
- Increment `count`
- Store `count` back into memory

Thus, when two threads execute the `count3s_thread()` code, these instructions might be interleaved, as shown in Figure 1.8. The result of the interleaved executions is that `count` is 1 rather than 2. Of course, many other interleavings are possible, some yielding correct results and others yielding incorrect results, but the fundamental problem is that the increment of `count` is not an *atomic operation*, that is, it is interruptible.

**Second Solution: Try 2.** We can solve this problem by using a *mutex* to provide *mutual exclusion*. A mutex is an object that has two states—locked and unlocked—and two methods—`lock()` and `unlock()`. The implementation of these methods ensures that when a thread attempts to lock a mutex, it checks to see if it is locked or unlocked. If it is locked, it waits until the mutex is in an unlocked state before locking it. By using a mutex to protect code that we wish to execute atomically—often...
referred to as a **critical section**—we guarantee that only one thread accesses it at any time. For the Count 3s problem, we simply lock a mutex before incrementing `count`, and we unlock the mutex after incrementing `count`, resulting in our second try at a solution (see Figure 1.9).

**Terminology.** Mutual exclusion and atomicity are related terms to describe an uninterrupted transformation.

**Mutual exclusion.** A piece of code executes with mutual exclusion if at most one thread can execute that code at any time.

**Atomicity.** The term atomicity comes from the database community, where a set of operations is atomic if either they all execute or none executes. Thus, there is no way to see the results of a partial execution.

With this modification, our second try is a correct parallel program. Unfortunately, as we can see from the graph shown in Figure 1.10, the solution is much slower than our original sequential code. With one thread, execution time is more than four times slower than the original serial code, so the overhead of using the mutexes is drastically harming performance. Worse, when we use two threads, each running on its own processor, our performance is even worse than with just one thread; here **lock contention** degrades performance, as each thread spends additional time waiting for the critical section to be unlocked.

**Third Solution: Try 3.** Recognizing the problem of lock overhead and lock contention, we implement a third version of our program that operates at a larger granularity or unit of sharing. Instead of accessing a critical section every time count

```c
1  mutex m;
2 3  void count3s_thread(int id) {
4   /* Compute portion of the array that this thread should work on */
5   int length_per_thread=length/t;
6   int start=id*length_per_thread;
7   8   for(i=start; i<start+length_per_thread; i++)
9     { 
10      if(array[i]==3)
11      { 
12       mutex_lock(m);
13       count++;
14       mutex_unlock(m);
15      } 
16     }
17  }
18 }
```

**Figure 1.9** The second try at a Count 3s solution showing the `count3s_thread()` with mutex protection for the `count` variable.
must be incremented, we can instead accumulate the local contribution to the overall count in a private variable, `private_count`, and only access the critical section for updating count once per thread. Our new code for this third solution is shown in Figure 1.11.

In exchange for a tiny amount of extra memory, our resulting program now executes considerably faster, as shown by the graph in Figure 1.12.

We see that with one thread our execution is close to the time of the serial code, so our latest changes have removed most of the locking overhead. However, with two threads there is still significant performance degradation. This time, the performance problem is more difficult to identify by simply inspecting the source code. We also need to understand some details of the underlying hardware. In particular, our hardware uses a protocol to maintain **coherent caches**, that is, to ensure that both processors “see” the same memory image. If processor 0 modifies a value at a given memory location, the hardware will invalidate any cached copy of that memory location that resides in processor 1’s L1 cache, thereby preventing processor 1 from accidentally accessing a stale value of the data. This cache coherence protocol becomes costly if two processors take turns repeatedly modifying the same data, because the data will repeatedly bounce between the two caches.

**Fourth Solution: Try 4.** In our code, there does not seem to be any shared modified data. However, the unit of cache coherence is known as a cache line, and for our machine the cache line size is 64 bytes. Thus, although the threads on processors P0 and P1 have exclusive access to either `private_count[0]` or `private_count[1]`, the underlying machine places them on the same 64 byte cache line; because cache coherence is maintained at the granularity of a cache line,
Parallelism Using Multiple Instruction Streams

Figure 1.11
The `count3s_thread()` for our third Count 3s solution using `private_count` array elements.

```c
private_count[MaxThreads];
mutex m;

void count3s_thread(int id)
{
  /* Compute portion of array for this thread to work on */
  int length_per_thread=length/t;
  int start=id*length_per_thread;

  for(i=start; i<start+length_per_thread; i++)
  {
    if(array[i] == 3)
    {
      private_count[id]++;
    }
  }
  mutex_lock(m);
  count+=private_count[id];
  mutex_unlock(m);
}
```

Figure 1.12
Performance results for our third Count 3s solution.
Chapter 1 Introduction

A modification of any part of a cache line is equivalent to a modification of the entire line, so this shared cache line bounces between the caches as `private_count[0]` and `private_count[1]` are repeatedly updated (see Figure 1.13). This phenomenon in which logically distinct data shares a physical cache line is known as false sharing. To eliminate false sharing, we can pad our array of private counters so that each resides on a distinct cache line (see Figure 1.14).

With this padding, the fourth solution removes both the overhead and contention of using mutexes, and we have largely achieved success, as shown in Figure 1.15. Our parallel solution running on one thread is almost as fast as the serial execution, the execution time of the parallel program is close to twice as fast when there are two threads, and it is almost four times as fast when there are four threads.

The only remaining problem with our solution is that with eight threads, the performance is no better than with four threads. There could be many possible reasons—related to the specifics of the hardware—for this behavior, but we conjecture that the L2 memory bandwidth is insufficient for such a large array. Figure 1.16 supports this conjecture by repeating the experiment on an array that does not contain any 3s. We see that even in this situation where there is no updating of shared variables, there is no performance improvement in moving from four to eight threads. While the Count 3s computation is unrealistic in the sense that it performs almost no work in relation to the amount of memory that it touches, this graph does point out one issue that will arise with future multi-core chips, namely, limited bandwidth to memory. Without changes in I/O technology, the bandwidth per core will shrink as increased transistor density supports a greater number of cores per chip without significantly increasing the chip perimeter on which I/O pins can be placed.

**Figure 1.13**
False Sharing. A cache line moves from RAM to the L3 cache, then to the L2 cache, and then to the L1 cache when a thread references its `private_count`. When the other thread references its `private_count`, the copy in the other L1 is invalidated, written back to the L2 cache, and then fetched into the other L1 cache. The line bounces between the L1 caches and the L2 cache, because although the references are to distinct memory locations, they use the same cache line.
1 struct padded_int
2 {
3    int value;
4    char padding[60];
5 } private_count[MaxThreads];
6
7 void count3s_thread(int id)
8 {
9    /* Compute portion of the array this thread should
10       work on */
11    int length_per_thread=length/t;
12    int start=id*length_per_thread;
13
14    for(i=start; i<start+length_per_thread; i++)
15    {
16      if(array[i] == 3)
17      {
18        private_count[id]++;
19      }
20    }
21    mutex_lock(m);
22    count+=private_count[id].value;
23    mutex_unlock(m);
24  }

Figure 1.14 The count3s_thread() function for our fourth solution to the Count 3s computations; the private count elements are padded to force them to be allocated to different cache lines.

Figure 1.15 Results for our fourth solution to the Count 3s problem shows that one processor has performance that is close to the sequential solution, that performance is almost twice as good with two processors and four times as good with four processors, but that eight processors provide no additional performance advantage.
Chapter 1  Introduction

Figure 1.16
Performance for our fourth solution to the Count 3s problem on an array that does not contain any 3s suggests that memory bandwidth limitations are preventing performance gains for eight processors.

Our Methodology. These results were obtained on an 8 processor Intel Xeon consisting of four 7100-series dual core microprocessor chips running at 2.60GHz. The system has three caches:

- **L3 cache:** 4 MB, unified, 16-way associative
  - 64B cache line size
- **L2 cache:** 1 MB (per core, total: 2 MB), unified, 8-way associative
- **L1 cache:** 16 KB data and 16 KB instruction, 8-way associative

The program operates on a 50M entry array that is randomly populated with 30% 3s and reports the average of 1000 program runs (including thread creation and destruction) using a microsecond timer. The results were obtained using GNU/Linux 2.6.19-gentoo-r5 and gcc version 4.1.2 with -O2 optimizations turned on.

From this example, we can see that producing correct and efficient parallel programs can be considerably more difficult than writing correct and efficient serial programs. The use of mutexes illustrated the need to control the interaction among processors carefully. The use of private counters illustrated the need to reason about the granularity of parallelism—that is, the frequency with which processes interact with one another. The use of padding showed the importance of understanding machine details, as sometimes small details can have large performance implications. It is this collection of interacting considerations that often makes parallel performance tuning difficult. Finally, we have seen two examples where we can trade off a small amount of memory for increased parallelism and increased performance.
The Goals: Scalability and Performance Portability

The Count 3s program illustrates both that performance can be achieved through parallelism and that achieving it can be complicated. Having mastered some of the issues facing multi-core processors—race conditions, issues of granularity, and false sharing—it’s tempting to think that parallel programming is concerned only with issues of correctness and performance. In fact, the goals of this book are broader. Our goal is to help you write good parallel programs, by which we mean parallel programs with four characteristics:

- They are correct
- They achieve good performance
- They are scalable to large numbers of processors
- They are portable across a wide variety of parallel platforms

The first goal does not require explanation, other than to notice that correctness can often be more difficult to achieve in a parallel program than in a sequential program. The second goal seems pretty clear, but as we will see in Chapter 3, defining what we mean by “good performance” is filled with subtleties.

The third and fourth goals, however, require some elaboration because they appear to be overly lofty and often unnecessary. For example, someone who programs a processor with a few cores has little interest in a parallel supercomputer with many thousands of processors. Indeed, there will always be some markets where the extreme desire for performance will dictate low-level non-portable solutions. But for the vast majority of programmers, scalability and portability are important because the landscape of parallel hardware is changing rapidly. For example, the first multi-core chips had only two cores per chip, but Intel has already discussed chips with 80 cores. Of course, as the number of cores increases, other micro-architectural features, such as the memory system, will have to change as well. Given this highly fluid hardware landscape, it’s best not to be caught scrambling when new hardware arrives. The solution is to design for scalability and portability from the beginning, so that programs will enjoy a long lifetime, justifying the significant intellectual and economic investment in their creation.

Let’s now briefly consider scalability and portability in more detail.

**Scalability**

To understand the issue of scalability, consider how our code is affected by increasing the number of processors. The Count 3s program was parameterized so that the number of threads could vary. This flexibility allows us to run the program on a sixteen-core chip with little modification. It would seem that we have produced a general solution that could scale to thousands simply by changing maxThreads. But
we have not. It’s true that the scan of the array, having been broken into segments, is independent, and therefore parallel for any number of threads. But the combining of the intermediate results is not, because all threads update one global sum. For a large number of threads, we would again encounter lock contention. Obviously, our pair-wise sum approach would fix this problem. Scalability requires scalable programming practices.

More generally, as the number of parallel processors increases, physical constraints force design changes that impact how programs perform. For example, communication latency—the delay encountered when transmitting information among processors—necessarily increases as the number of processors grows simply because of speed of light limitations. On a single chip, different issues apply, but they still affect communication latency when the number of cores grows large. For a small number of processors, proximity allows certain operations to be fast, but these operations do not remain fast as the size of the system grows. Exploiting these benefits makes sense when possible, but the program must avoid relying on them for its success. Well-written parallel programs can exploit the fast components and avoid over-using the slow components of a parallel computer.

**Performance Portability**

The problem just discussed—that physical constraints impact the characteristics of parallel computers as the number of processors increases—is not limited to slowing down certain operations. The problem is much more insidious.

Architects, grappling with physical constraints, have created scores of parallel computer designs. These machines can differ dramatically from each other. Unlike the sequential case, where a new computer usually requires only a recompilation of the source code to execute respectably well, a program running well on one parallel machine may have to be rewritten for a new one.

To give one example, parallel computers can mostly be divided into one of three classes: shared memory, typified by multi-core processors, shared address space, typified by various supercomputers, and separately addressed memories (shared nothing), typified by clusters. This distinction affects every memory reference in a program, so it has a tremendous impact on how the program should be written. Programs intended to port to all of these platforms must be robust to these differences in memory structure, and techniques for ensuring robustness will concern us throughout the book.

The classification by memory capability specifies the variety along one axis. There are many other differences among parallel processors. We could solve the portability problem by simply setting a high enough level of abstraction that none of these differences is visible; then, a compiler will map the high level specification to the platform. The strategy will make our programs insensitive to the parallel hardware, but it’s not a good idea. Generally, though compilers can perform the mapping, they will usually introduce software layers to implement the abstractions; the added software
hides the performance features of the hardware, making it difficult for programmers to know how their code behaved. We cannot divorce ourselves entirely from the underlying hardware if we want high performance. So, we will use a different strategy, described in Chapter 2.

Our goal, then, is portability with performance, often called performance portability. It’s not enough for the program to run on different parallel machines. It must run well on all of them.

**Principles First**

This book does not provide a step-by-step tutorial for writing good parallel programs. Instead, it emphasizes the principles underlying parallel computation, explaining the various phenomena and explaining why they represent opportunities or barriers to successful parallel programming. Our reasons for this approach are twofold. First, by focusing on principles, we hope to provide enduring knowledge that will outlive the specifics of the latest hardware or software technology, which as we’ve pointed out, change rapidly. Second, and more importantly, the parallel programming community does not yet have all of the answers, so a step-by-step solution is not available. Indeed, one of our goals is to inspire the next generation of researchers to understand the limitations of current technology so that they can build the better solutions of tomorrow.

After presenting these principles, we discuss some popular programming languages and tools used for programming contemporary parallel machines. Again, our goal is more concerned with the principles behind the approaches than with turning the reader into an expert in a specific language. Our treatments, therefore, are minimal, and readers should expect to consult reference manuals for more complete and detailed information.

**Chapter Summary**

The chapter began with the observation that parallelism—doing two or more things at once to achieve a single goal—is a familiar idea that we encounter in everyday life. Though familiar, parallelism has not been a significant aspect of programming in the past because sequential computer performance has increased steadily for decades. Such improvements have been due to a combination of technology improvements and the incorporation of parallelism into sequential processor design by computer architects (hidden parallelism). Because the architectural opportunities have largely been mined, the continued advancement of technology has made computers with multiple processors standard. This shift is having a profound effect on computer programming.

We noted that existing sequential programs generally cannot take advantage of a parallel computer. The primary reason is that existing programming languages and standard programming techniques strongly incorporate the sequential processing
of the traditional von Neumann computer architecture. Parallel solutions, as illustrated by several simple computations—summation, parallel prefix and Count 3s—illustrated features of parallel computations. Though they might not have been the first solutions to come to mind, they were still quite intuitive. A change in thinking about computation will be required—we called it a shift in paradigm—before programmers instinctively devise parallel solutions to their computational problems.

In a quick and incomplete survey of parallel hardware, we noted platforms as diverse as chips with two processors to server centers with thousands of processors. Though dramatically different in scale and design, their parallel features rely on a small set of fundamental principles. We committed to focusing on those principles with the goal of empowering programmers to strive for parallel programs that achieve high performance, scalability, and performance-portability.

**Historical Perspective**

Parallelism has been applied in the design of sequential computers since the first commercial machines in the 1950s. A landmark parallel machine was the Illiac IV, built in the 1970s by a team at the University of Illinois, Urbana-Champaign. Though the Illiac IV was successfully programmed in low-level assembly-like code, the task of developing a compiler to translate sequential (Fortran) programs into a parallel form was begun by David Kuck and colleagues. Investigators throughout the community pursued the goal to the end of the century, resulting in an enormous literature on parallelizing compilers.

**Exercises**

1. Explain the meaning of the following vocabulary related to thread programming:
   a. Thread
   b. Race Condition
   c. Mutex
   d. Lock Contention
   e. Granularity
   f. False Sharing

2. Describe how the pair-wise summation computation can be changed to find the maximum element of an array.

3. Reformulate the pair-wise summation program to solve the Count 3s computation in \( \log n \) time, assuming \( P = n/2 \).

4. Reformulate the pair-wise summation program to solve the Count 3s computation assuming that \( n = 1024 \), but \( P = 8 \).

5. Rewrite the iterative summation program using `for all`; don’t forget about race conditions.
6. Locate the closest parallel computer—in your laptop, perhaps, or in a lab—and find out how many processors it has, how much memory each processor can access, and what languages and software are available to program it. Write a “hello world” program for this computer.

7. As presented, the tree summation algorithm was always illustrated with $n = 2^m$, causing the tree to be perfectly balanced. Revise the algorithm for the case when $n$ is not a power of 2.

8. As presented, the tree summation algorithm requires $P = n/2$ processors, which allows it to achieve full parallelism. Revise the tree summation computation to work with fewer processors.

9. Write any sequence of 16 integers. Create a new sequence of “max prefixes” using the parallel prefix algorithm; build the tree structure (Figure 1.4) and construct the upward and downward value flows. What value “flows into the root” if the algorithm is to work with signed numbers?

10. Write the C code segments executed at the nodes in the parallel prefix algorithm for a) the upward flow, and b) the downward flow.