CHAPTER

# Introduction

# 1

At the dawn of the twenty-first century, the computer industry underwent yet another revolution. The major chip manufacturers had increasingly been unable to make processor chips both smaller and faster. As Moore's law approached the end of its 50-year reign, manufacturers turned to "multicore" architectures, in which multiple processors (cores) on a single chip communicate directly through shared hardware caches. Multicore chips make computing more effective by exploiting *parallelism*: harnessing multiple circuits to work on a single task.

The spread of multiprocessor architectures has had a pervasive effect on how we develop software. During the twentieth century, advances in technology brought regular increases in clock speed, so software would effectively "speed up" by itself over time. In this century, however, that "free ride" has come to an end. Today, advances in technology bring regular increases in parallelism, but only minor increases in clock speed. Exploiting that parallelism is one of the outstanding challenges of modern computer science.

This book focuses on how to program multiprocessors that communicate via a shared memory. Such systems are often called *shared-memory multiprocessors* or, more recently, *multicores*. Programming challenges arise at all scales of multiprocessor systems—at a very small scale, processors within a single chip need to coordinate access to a shared memory location, and on a large scale, processors in a supercomputer need to coordinate the routing of data. Multiprocessor programming is challenging because modern computer systems are inherently *asynchronous*: activities can be halted or delayed without warning by interrupts, preemption, cache misses, failures, and other events. These delays are inherently unpredictable, and can vary enormously in scale: a cache miss might delay a processor for fewer than ten instructions, a page fault for a few million instructions, and operating system preemption for hundreds of millions of instructions.

We approach multiprocessor programming from two complementary directions: principles and practice. In the *principles* part of this book, we focus on *computability*: figuring out what can be computed in an asynchronous concurrent environment. We use an idealized model of computation in which multiple concurrent *threads* manipulate a set of shared *objects*. The sequence of the thread operations on the objects is called the *concurrent program* or *concurrent algorithm*. This model is essentially the model presented by threads in Java, C#, and C++.

Surprisingly, there are easy-to-specify shared objects that cannot be implemented by any concurrent algorithm. It is therefore important to understand what not to try, before proceeding to write multiprocessor programs. Many of the issues that will land multiprocessor programmers in trouble are consequences of fundamental limitations of the computational model, so we view the acquisition of a basic understanding of concurrent shared-memory computability as a necessary step. The chapters dealing with principles take the reader through a quick tour of asynchronous computability, attempting to expose various computability issues, and how they are addressed through the use of hardware and software mechanisms.

An important step in the understanding of computability is the specification and verification of what a given program actually does. This is perhaps best described as *program correctness*. The correctness of multiprocessor programs, by their very nature, is more complex than that of their sequential counterparts, and requires a different set of tools, even for the purpose of "informal reasoning" (which, of course, is what most programmers actually do).

Sequential correctness is mostly concerned with safety properties. A *safety* property states that some "bad thing" never happens. For example, a traffic light never displays green in all directions, even if the power fails. Naturally, concurrent correctness is also concerned with safety, but the problem is much, much harder, because safety must be ensured despite the vast number of ways that the steps of concurrent threads can be interleaved. Equally important, concurrent correctness encompasses a variety of *liveness* properties that have no counterparts in the sequential world. A *liveness* property states that a particular good thing will happen. For example, a red traffic light will eventually turn green.

A final goal of the part of the book dealing with principles is to introduce a variety of metrics and approaches for reasoning about concurrent programs, which will later serve us when discussing the correctness of real-world objects and programs.

The second part of the book deals with the *practice* of multiprocessor programming, and focuses on performance. Analyzing the performance of multiprocessor algorithms is also different in flavor from analyzing the performance of sequential programs. Sequential programming is based on a collection of well-established and well-understood abstractions. When we write a sequential program, we can often ignore that underneath it all, pages are being swapped from disk to memory, and smaller units of memory are being moved in and out of a hierarchy of processor caches. This complex memory hierarchy is essentially invisible, hiding behind a simple programming abstraction.

In the multiprocessor context, this abstraction breaks down, at least from a performance perspective. To achieve adequate performance, programmers must sometimes "outwit" the underlying memory system, writing programs that would seem bizarre to someone unfamiliar with multiprocessor architectures. Someday, perhaps, concurrent architectures will provide the same degree of efficient abstraction as sequential architectures, but in the meantime, programmers should beware.

The practice part of the book presents a progressive collection of shared objects and programming tools. Every object and tool is interesting in its own right, and we use each one to expose the reader to higher-level issues: spin locks illustrate contention, linked lists illustrate the role of locking in data structure design, and so on.

Each of these issues has important consequences for program performance. We hope that readers will understand the issue in a way that will later allow them to apply the lessons learned to specific multiprocessor systems. We culminate with a discussion of state-of-the-art technologies such as *transactional memory*.

For most of this book, we present code in the Java programming language, which provides automatic memory management. However, memory management is an important aspect of programming, especially concurrent programming. So, in the last two chapters, we switch to C++. In some cases, the code presented is simplified by omitting nonessential details. Complete code for all the examples is available on the book's companion website at *https://textbooks.elsevier.com/web/product_details.aspx?isbn=978124159501*.

There are, of course, other languages which would have worked as well. In the appendix, we explain how the concepts expressed here in Java or C++ can be expressed in some other popular languages or libraries. We also provide a primer on multiprocessor hardware.

Throughout the book, we avoid presenting specific performance numbers for programs and algorithms, instead focusing on general trends. There is a good reason why: multiprocessors vary greatly, and what works well on one machine may work significantly less well on another. We focus on general trends to ensure that observations are not tied to specific platforms at specific times.

Each chapter has suggestions for further reading, along with exercises suitable for Sunday morning entertainment.

## 1.1 Shared objects and synchronization

On the first day of your new job, your boss asks you to find all primes between 1 and $10^{10}$ (never mind why) using a parallel machine that supports ten concurrent threads. This machine is rented by the minute, so the longer your program takes, the more it costs. You want to make a good impression. What do you do?

As a first attempt, you might consider giving each thread an equal share of the input domain. Each thread might check $10^9$ numbers, as shown in Fig. 1.1. This

```
void primePrint {
  int i = ThreadID.get();   // thread IDs are in {0..9}
  long block = power(10, 9);
  for (long j = (i * block) + 1; j <= (i + 1) * block; j++) {
    if (isPrime(j))
      print(j);
  }
}
```

**FIGURE 1.1**

Balancing the work load by dividing up the input domain. Each thread in {0..9} gets an equal subset of the range.

```
Counter counter = new Counter(1);     // shared by all threads
void primePrint {
  long i = 0;
  long limit = power(10, 10);
  while (i < limit) {                 // loop until all numbers taken
    i = counter.getAndIncrement();    // take next untaken number
    if (isPrime(i))
      print(i);
  }
}
```

**FIGURE 1.2**

Balancing the work load using a shared counter. Each thread is given a dynamically determined number of numbers to test.

```
public class Counter {
  private long value;               // initialized by constructor
  public Counter(long i) {
    value = i;
  }
  public long getAndIncrement() { // increment, returning prior value
    return value++;
  }
}
```

**FIGURE 1.3**

An implementation of the shared counter.

approach fails to distribute the work evenly for an elementary but important reason: Equal ranges of inputs do not produce equal amounts of work. Primes do not occur uniformly; there are more primes between 1 and $10^9$ than between $9 \cdot 10^9$ and $10^{10}$. To make matters worse, the computation time per prime is not the same in all ranges: it usually takes longer to test whether a large number is prime than a small number. In short, there is no reason to believe that the work will be divided equally among the threads, and it is not clear even which threads will have the most work.

A more promising way to split the work among the threads is to assign each thread one integer at a time (Fig. 1.2). When a thread is finished testing an integer, it asks for another. To this end, we introduce a *shared counter*, an object that encapsulates an integer value, and that provides a `getAndIncrement()` method, which increments the counter's value and returns the counter's prior value.

Fig. 1.3 shows a naïve implementation of `Counter` in Java. This counter implementation works well when used by a single thread, but it fails when shared by multiple threads. The problem is that the expression

```
return value++;
```

is in effect an abbreviation of the following, more complex code:

```
long temp = value;
value = temp + 1;
return temp;
```

In this code fragment, `value` is a field of the `Counter` object, and is shared among all the threads. Each thread, however, has its own copy of `temp`, which is a local variable to each thread.

Now imagine that two threads call the counter's `getAndIncrement()` method at about the same time, so that they both read 1 from `value`. In this case, each thread would set its local `temp` variables to 1, set `value` to 2, and return 1. This behavior is not what we intended: we expect concurrent calls to the counter's `getAndIncrement()` to return distinct values. It could be worse: after one thread reads 1 from `value`, but before it sets `value` to 2, another thread could go through the increment loop several times, reading 1 and writing 2, then reading 2 and writing 3. When the first thread finally completes its operation and sets `value` to 2, it will actually be setting the counter back from 3 to 2.

The heart of the problem is that incrementing the counter's value requires two distinct operations on the shared variable: reading the `value` field into a temporary variable and writing it back to the `Counter` object.

Something similar happens when you try to pass someone approaching you head-on in a corridor. You may find yourself veering right and then left several times to avoid the other person doing exactly the same thing. Sometimes you manage to avoid bumping into them and sometimes you do not. In fact, as we will see in the later chapters, such collisions are provably unavoidable.[1] On an intuitive level, what is going on is that each of you is performing two distinct steps: looking at ("reading") the other's current position, and moving ("writing") to one side or the other. The problem is, when you read the other's position, you have no way of knowing whether they have decided to stay or move. In the same way that you and the annoying stranger must decide on which side to pass each other, threads accessing a shared `Counter` must decide who goes first and who goes second.

As we discuss in Chapter 5, modern multiprocessor hardware provides special *read–modify–write* instructions that allow threads to read, modify, and write a value to memory in one *atomic* (that is, indivisible) hardware step. For the `Counter` object, we can use such hardware to increment the counter atomically.

We can also ensure atomic behavior by guaranteeing in software (using only read and write instructions) that only one thread executes the read-and-write sequence at a time. The problem of ensuring that only one thread can execute a particular block of code at a time, called the *mutual exclusion* problem, is one of the classic coordination problems in multiprocessor programming.

[1] A preventive approach such as "always sidestep to the right" does not work because the approaching person may be British.

As a practical matter, you are unlikely ever to find yourself having to design your own mutual exclusion algorithm (you would probably call on a library). Nevertheless, understanding how to implement mutual exclusion from the basics is an essential condition for understanding concurrent computation in general. There is no more effective way to learn how to reason about essential and ubiquitous issues such as mutual exclusion, deadlock, bounded fairness, and blocking versus nonblocking synchronization.

## 1.2 A fable

Instead of treating coordination problems (such as mutual exclusion) as programming exercises, we prefer to frame concurrent coordination problems as interpersonal problems. In the next few sections, we present a sequence of fables, illustrating some of the basic problems. Like most authors of fables, we retell stories mostly invented by others (see the chapter notes at the end of this chapter).

Alice and Bob are neighbors, and they share a yard. Alice owns a cat and Bob owns a dog. Both pets like to run around in the yard, but (naturally) they do not get along. After some unfortunate experiences, Alice and Bob agree that they should coordinate to make sure that both pets are never in the yard at the same time. Of course, they rule out trivial solutions that do not allow either pet into an empty yard, or that reserve the yard exclusively to one pet or the other.

How should they do it? Alice and Bob need to agree on mutually compatible procedures for deciding what to do. We call such an agreement a *coordination protocol* (or just a *protocol*, for short).

The yard is large, so Alice cannot simply look out of the window to check whether Bob's dog is present. She could perhaps walk over to Bob's house and knock on the door, but that takes a long time, and what if it rains? Alice might lean out the window and shout "Hey Bob! Can I let the cat out?" The problem is that Bob might not hear her. He could be watching TV, visiting his girlfriend, or out shopping for dog food. They could try to coordinate by cell phone, but the same difficulties arise if Bob is in the shower, driving through a tunnel, or recharging his phone's batteries.

Alice has a clever idea. She sets up one or more empty beer cans on Bob's windowsill (Fig. 1.4), ties a string around each one, and runs the string back to her house. Bob does the same. When she wants to send a signal to Bob, she yanks the string to knock over one of the cans. When Bob notices a can has been knocked over, he resets the can.

Up-ending beer cans by remote control may seem like a creative idea, but it does not solve this problem. The problem is that Alice can place only a limited number of cans on Bob's windowsill, and sooner or later, she is going to run out of cans to knock over. Granted, Bob resets a can as soon as he notices it has been knocked over, but what if he goes to Cancún for spring break? As long as Alice relies on Bob to reset the beer cans, sooner or later, she might run out.