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.

then we end up with a five-fold speedup, not a 10-fold speedup. In other words, the remaining 10% that we did not parallelize cut our utilization of the machine in half. It seems worthwhile to invest effort to derive as much parallelism from the remaining 10% as possible, even if it is difficult. Typically, it is hard because these additional parallel parts involve substantial communication and coordination. A major focus of this book is understanding the tools and techniques that allow programmers to effectively program the parts of the code that require coordination and synchronization, because the gains made on these parts may have a profound impact on performance.

Returning to the prime number printing program of Fig. 1.2, let us revisit the three main lines of code:

```
i = counter.getAndIncrement(); // take next untaken number
if (isPrime(i))
  print(i);
```

It would have been simpler to have threads perform these three lines atomically, that is, in a single mutually exclusive block. Instead, only the call to `getAndIncrement()` is atomic. This approach makes sense when we consider the implications of Amdahl's law: It is important to minimize the granularity of sequential code, in this case, the code accessed using mutual exclusion. Moreover, it is important to implement mutual exclusion in an effective way, since the communication and coordination around the mutually exclusive shared counter can substantially affect the performance of our program as a whole.

1.6 Parallel programming

For many of the applications we wish to parallelize, significant parts can easily be determined as executable in parallel because they do not require any form of coordination or communication. However, at the time this book is being written, there is no cookbook recipe for identifying these parts. This is where the application designer must use his or her accumulated understanding of the algorithm being parallelized. Luckily, in many cases it is obvious how to identify such parts. The more substantial problem, the one which this book addresses, is how to deal with the remaining parts of the program. As noted earlier, these are the parts that cannot be parallelized easily because the program must access shared data and requires interprocess coordination and communication in an essential way.

The goal of this text is to expose the reader to core ideas behind modern coordination paradigms and concurrent data structures. We present the reader with a unified, comprehensive picture of the elements that are key to effective multiprocessor programming, ranging from basic principles to best-practice engineering techniques.

Multiprocessor programming poses many challenges, ranging from grand intellectual issues to subtle engineering tricks. We tackle these challenges using successive refinement, starting with an idealized model in which mathematical concerns are paramount, and gradually moving on to more pragmatic models, where we increasingly focus on basic engineering principles.

For example, the first problem we consider is mutual exclusion, the oldest and still one of the fundamental problems in the field. We begin with a mathematical perspective, analyzing the computability and correctness properties of various algorithms on an idealized architecture. The algorithms themselves, while classical, are not practical for modern multicore architectures. Nevertheless, learning how to reason about such idealized algorithms is an important step toward learning how to reason about more realistic (and more complex) algorithms. It is particularly important to learn how to reason about subtle liveness issues such as starvation and deadlock.

Once we understand how to reason about such algorithms in general, we turn our attention to more realistic contexts. We explore a variety of algorithms and data structures using different multiprocessor architectures with the goal of understanding which are effective, and why.

1.7 Chapter notes

Most of the parable of Alice and Bob is adapted from Leslie Lamport's invited lecture at the 1984 ACM Symposium on Principles of Distributed Computing [104]. The readers–writers problem is a classical synchronization problem that has received attention in numerous papers over the past 20 years. Amdahl's law is due to Gene Amdahl, a parallel processing pioneer [9].

1.8 Exercises

Exercise 1.1. The *dining philosophers problem* was invented by E.W. Dijkstra, a concurrency pioneer, to clarify the notions of deadlock- and starvation-freedom. Imagine five philosophers who spend their lives just thinking and feasting on rice. They sit around a circular table, illustrated in Fig. 1.5. However, there are only five chopsticks

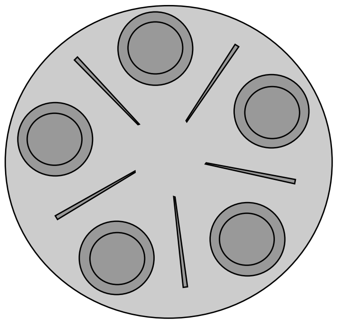

FIGURE 1.5

Traditional dining table arrangement according to Dijkstra.

APPENDIX

Hardware basics

B

A novice was trying to fix a broken Lisp machine by turning the power off and on. Knight, seeing what the student was doing spoke sternly: "You cannot fix a machine just by power-cycling it with no understanding of what is going wrong." Knight turned the machine off and on. The machine worked.

(From "AI Koans," a collection of jokes popular at MIT in the 1980s.)

B.1 Introduction (and a puzzle)

You can do a pretty good job of programming a uniprocessor without understanding much about computer architecture, but the same is not true of multiprocessors. You cannot program a multiprocessor effectively unless you know what a multiprocessor *is*. We illustrate this point by a puzzle. We consider two programs that are logically equivalent, but one is much less efficient than the other. Ominously, the simpler program is the inefficient one. This discrepancy cannot be explained, nor the danger avoided, without a basic understanding of modern multiprocessor architectures.

Here is the background to the puzzle. Suppose two threads share a resource that can be used by only one thread at a time. To prevent concurrent use, each thread must *lock* the resource before using it, and *unlock* it afterward. We study many ways to implement locks in Chapter 7. For the puzzle, we consider two simple implementations in which the lock is a single Boolean field. If the field is *false*, the lock is free, and otherwise it is in use. We manipulate the lock with the `getAndSet`(v) method, which atomically swaps its argument v with the field value. To acquire the lock, a thread calls `getAndSet`(*true*). If the call returns *false*, then the lock was free, and the caller succeeded in locking the object. Otherwise, the object was already locked, and the thread must try again later. A thread releases a lock simply by storing *false* into the Boolean field.

In Fig. B.1, the *test-and-set* (`TASLock`) lock repeatedly calls `getAndSet`(*true*) (line 4) until it returns *false*. By contrast, in Fig. B.2, the *test-and-test-and-set* lock (`TTASLock`) repeatedly reads the lock field (by calling `state.get()` at line 5) until it returns *false*, and only then calls `getAndSet()` (line 6). It is important to understand that reading the lock value is atomic, and applying `getAndSet()` to the lock value is atomic, but the combination is not atomic: Between the time a thread reads the lock value and the time it calls `getAndSet()`, the lock value may have changed.

```
public class TASLock implements Lock {
  ...
  public void lock() {
    while (state.getAndSet(true)) {} // spin
  }
  ...
}
```

FIGURE B.1

The `TASLock` class.

```
public class TTASLock implements Lock {
  ...
  public void lock() {
    while (true) {
      while (state.get()) {}; // spin
      if (!state.getAndSet(true))
        return;
    }
  }
  ...
}
```

FIGURE B.2

The `TTASLock` class.

Before you proceed, you should convince yourself that the `TASLock` and `TTASLock` algorithms are logically the same. The reason is simple: In the `TTASLock` algorithm, reading that the lock is free does not guarantee that the next call to `getAndSet()` will succeed, because some other thread may have acquired the lock in the interval between reading the lock and trying to acquire it. So why bother reading the lock before trying to acquire it?

Here is the puzzle: While the two lock implementations may be logically equivalent, they perform very differently. In a classic 1989 experiment, Anderson measured the time needed to execute a simple test program on several contemporary multiprocessors. He measured the elapsed time for n threads to execute a short critical section one million times. Fig. B.3 shows how long each lock takes, plotted as a function of the number of threads. In a perfect world, both the `TASLock` and `TTASLock` curves would be as flat as the ideal curve on the bottom, since each run does the same number of increments. Instead, we see that both curves slope up, indicating that lock-induced delay increases with the number of threads. Curiously, however, the `TASLock` is much slower than the `TTASLock` lock, especially as the number of threads increases. Why?

This appendix covers much of what you need to know about multiprocessor architecture to write efficient concurrent algorithms and data structures. Along the way, we will explain the divergent curves in Fig. B.3.

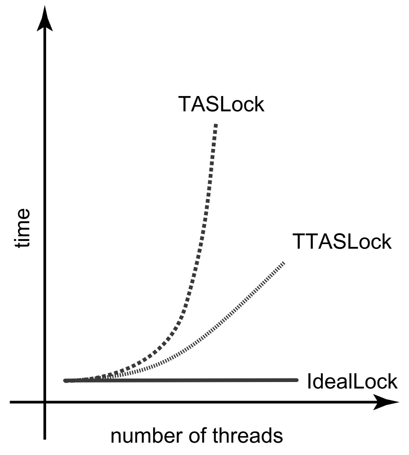

FIGURE B.3

Schematic performance of a `TASLock`, a `TTASLock`, and an ideal lock.

We are concerned with the following components:

- The *processors* are hardware devices that execute software *threads*. There are typically more threads than processors, and each processor runs a thread for a while, sets it aside, and turns its attention to another thread.
- The *interconnect* is a communication medium that links processors to processors and processors to memory.
- The *memory* is actually a hierarchy of components that store data, ranging from one or more levels of small, fast *caches* to a large and relatively slow *main memory*. Understanding how these levels interact is essential to understanding the actual performance of many concurrent algorithms.

From our point of view, one architectural principle drives everything else: *Processors and main memory are far apart*. It takes a long time for a processor to read a value from memory. It also takes a long time for a processor to write a value to memory, and longer still for the processor to be sure that value has actually been installed in memory. Accessing memory is more like mailing a letter than making a phone call. Almost everything we examine in this appendix is the result of trying to alleviate the long time it takes ("high latency") to access memory.

Processor and memory speed change over time, but their *relative* performance changes slowly. Consider the following analogy. Imagine that it is 1980, and you are in charge of a messenger service in mid-town Manhattan. While cars outperform bicycles on the open road, bicycles outperform cars in heavy traffic, so you choose to use bicycles. Even though the technology behind both bicycles and cars has advanced, the *architectural* comparison remains the same. Then, as now, if you are designing an urban messenger service, you should use bicycles, not cars.

B.2 Processors and threads

A multiprocessor consists of multiple hardware *processors*, each of which executes a sequential program. When discussing multiprocessor architectures, the basic unit of time is the *cycle*: the time it takes a processor to fetch and execute a single instruction. In absolute terms, cycle times change as technology advances (from about 10 million cycles per second in 1980 to about 3000 million in 2005), and they vary from one platform to another (processors that control toasters have longer cycles than processors that control web servers). Nevertheless, the relative cost of instructions such as memory access changes slowly when expressed in terms of cycles.

A *thread* is a sequential program. While a processor is a hardware device, a thread is a software construct. A processor can run a thread for a while and then set it aside and run another thread, an event known as a *context switch*. A processor may set aside a thread, or *deschedule* it, for a variety of reasons. Perhaps the thread has issued a memory request that will take some time to satisfy, or perhaps that thread has simply run long enough, and it is time for another thread to make progress. When a thread is descheduled, it may resume execution on another processor.

B.3 Interconnect

The *interconnect* is the medium by which processors communicate with the memory and with other processors. There are essentially two kinds of interconnect architectures in use: *symmetric multiprocessing* (SMP) and *nonuniform memory access* (NUMA). See Fig. B.4.

In an SMP architecture, processors and memory are linked by a *bus* interconnect, a broadcast medium that acts like a tiny ethernet. Both processors and the main memory have *bus controller* units in charge of sending and listening for messages broadcast on the bus. (Listening is sometimes called *snooping*.) SMP architectures are easier to build, but they are not scalable to large numbers of processors because eventually the bus becomes overloaded.

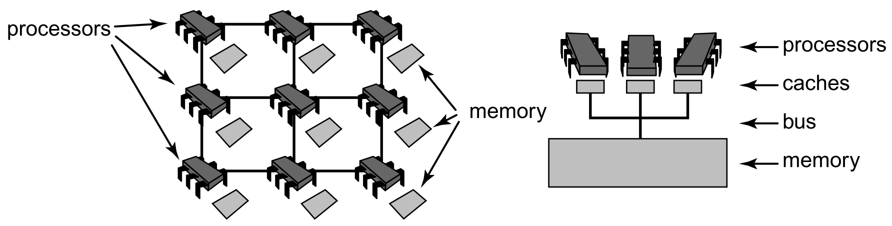

FIGURE B.4

An SMP architecture with caches on the right and a cacheless NUMA architecture on the left.

In a NUMA architecture, a collection of *nodes* are linked by a point-to-point network, like a tiny local area network. Each node contains one or more processors and a local memory. One node's local memory is accessible to the other nodes, and together, the nodes' memories form a global memory shared by all processors. The NUMA name reflects the fact that a processor can access memory residing on its own node faster than it can access memory residing on other nodes. Networks are more complex than buses, and require more elaborate protocols, but they scale better than buses to large numbers of processors.

The division between SMP and NUMA architectures is a simplification: For example, some systems have hybrid architectures, where processors within a cluster communicate over a bus, but processors in different clusters communicate over a network.

From the programmer's point of view, it may not seem important whether the underlying platform is based on a bus, a network, or a hybrid interconnect. It is important, however, to realize that the interconnect is a finite resource shared among the processors. If one processor uses too much of the interconnect's bandwidth, then the others may be delayed.

B.4 Memory

Processors share a *main memory*, which is a large array of *words*, indexed by *address*. The size of a word or an address is platform-dependent, but typically it is either 32 or 64 bits. Simplifying somewhat, a processor reads a value from memory by sending a message containing the desired address to memory. The response message contains the associated *data*, that is, the contents of memory at that address. A processor writes a value by sending the address and the new data to memory, and the memory sends back an acknowledgment when the new data have been installed.

B.5 Caches

On modern architectures, a main memory access may take hundreds of cycles, so there is a real danger that a processor may spend much of its time just waiting for the memory to respond to requests. Modern systems alleviate this problem by introducing one or more *caches*: small memories that are situated closer to the processors and are therefore much faster than main memory. These caches are logically situated "between" the processor and the memory: When a processor attempts to read a value from a given memory address, it first looks to see if the value is already in the cache, and if so, it does not need to perform the slower access to memory. If the desired address's value was found, we say the processor *hits* in the cache, and otherwise it *misses*. In a similar way, if a processor attempts to write an address that is in the cache, it does not need to perform the slower access to memory. The proportion of requests satisfied in the cache is called the cache *hit ratio* (or *hit rate*).

Caches are effective because most programs display a high degree of *locality*: If a processor reads or writes a memory address (also called a memory location), then it is likely to read or write the same location again soon. Moreover, if a processor reads or writes a memory location, then it is also likely to read or write *nearby* locations soon. To exploit this second observation, caches typically operate at a *granularity* larger than a single word: A cache holds a group of neighboring words called *cache lines* (sometimes called *cache blocks*).

In practice, most processors have two or three levels of caches, called $L1$, $L2$, and $L3$ caches. All but the last (and largest) cache typically reside on the same chip as the processor. An L1 cache typically takes one or two cycles to access. An on-chip L2 may take about 10 cycles to access. The last level cache, whether L2 or L3, typically takes tens of cycles to access. These caches are significantly faster than the hundreds of cycles required to access the memory. Of course, these times vary from platform to platform, and many multiprocessors have even more elaborate cache structures.

The original proposals for NUMA architectures did not include caches because it was felt that local memory was enough. Later, however, commercial NUMA architectures did include caches. Sometimes the term *cache-coherent NUMA* (cc-NUMA) is used to mean NUMA architectures with caches. Here, to avoid ambiguity, we use NUMA to include cache-coherence unless we explicitly state otherwise.

Caches are expensive to build and therefore significantly smaller than the memory: Only a fraction of the memory locations will fit in a cache at the same time. We would therefore like the cache to maintain values of the most highly used locations. This implies that when a location needs to be cached and the cache is full, it is necessary to *evict* a line, discarding it if it has not been modified, and writing it back to main memory if it has. A *replacement policy* determines which cache line to replace to make room for a given new location. If the replacement policy is free to replace any line, then we say the cache is *fully associative*. If, on the other hand, there is only one line that can be replaced, then we say the cache is *direct-mapped*. If we split the difference, allowing any line from a *set* of size k to be replaced to make room for a given line, then we say the cache is *k-way set associative*.

B.5.1 Coherence

Sharing (or, less politely, *memory contention*) occurs when one processor reads or writes a memory address that is cached by another. If both processors are reading the data without modifying it, then the data can be cached at both processors. If, however, one processor tries to update the shared cache line, then the other's copy must be *invalidated* to ensure that it does not read an out-of-date value. In its most general form, this problem is called *cache-coherence*. The literature contains a variety of very complex and clever cache coherence protocols. Here we review one of the most commonly used, called the *MESI* protocol (pronounced "messy") after the names of possible cache line states. (Modern processors tend to use more complex protocols with additional states.) Here are the cache line states:

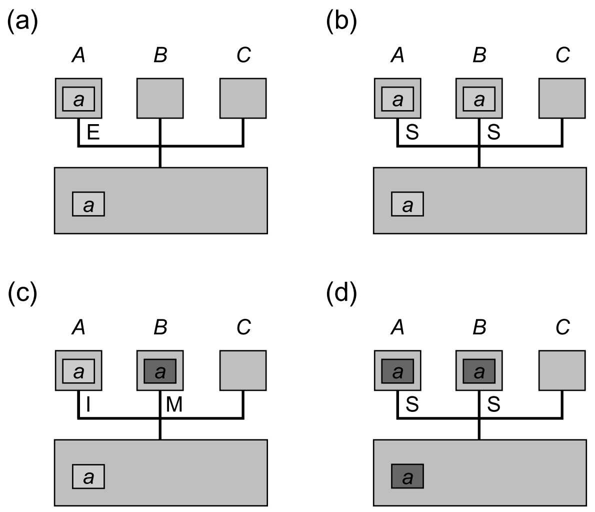

FIGURE B.5

Example of the MESI cache-coherence protocol's state transitions. (a) Processor *A* reads data from address *a*, and stores the data in its cache in the *exclusive* state. (b) When processor *B* attempts to read from the same address, *A* detects the address conflict, and responds with the associated data. Now *a* is cached at both *A* and *B* in the *shared* state. (c) If *B* writes to the shared address *a*, it changes its state to *modified*, and broadcasts a message warning *A* (and any other processor that might have those data cached) to set its cache line state to *invalid*. (d) If *A* then reads from *a*, it broadcasts a request, and *B* responds by sending the modified data both to *A* and to the main memory, leaving both copies in the *shared* state.

- *Modified*: The line has been modified in the cache, and it must eventually be written back to main memory. No other processor has this line cached.
- *Exclusive*: The line has not been modified, and no other processor has this line cached.
- *Shared*: The line has not been modified, and other processors may have this line cached.
- *Invalid*: The line does not contain meaningful data.

We illustrate this protocol by a short example depicted in Fig. B.5. For simplicity, we assume processors and memory are linked by a bus.

Processor *A* reads data from address *a*, and stores the data in its cache in the *exclusive* state. When processor *B* attempts to read from the same address, *A* detects the address conflict, and responds with the associated data. Now *a* is cached at both *A* and *B* in the *shared* state. If *B* writes to the shared address *a*, it changes its state to *modified*, and broadcasts a message warning *A* (and any other processor that might have those data cached) to set its cache line state to *invalid*. If *A* then reads from *a*, it broadcasts a request, and *B* responds by sending the modified data both to *A* and to the main memory, leaving both copies in the *shared* state.

False sharing occurs when processors that are accessing logically distinct data nevertheless conflict because the locations they are accessing lie on the same cache line. This observation illustrates a difficult trade-off: Large cache lines are good for locality, but they increase the likelihood of false sharing. The likelihood of false sharing can be reduced by ensuring that data objects that might be accessed concurrently by independent threads lie far enough apart in memory. For example, having multiple threads share a byte array invites false sharing, but having them share an array of double-precision integers is less dangerous.

B.5.2 Spinning

A processor is *spinning* if it is repeatedly testing some word in memory, waiting for another processor to change it. Depending on the architecture, spinning can have a dramatic effect on overall system performance.

On an SMP architecture without caches, spinning is a very bad idea. Each time the processor reads the memory, it consumes bus bandwidth without accomplishing any useful work. Because the bus is a broadcast medium, these requests directed to memory may prevent other processors from making progress.

On a NUMA architecture without caches, spinning may be acceptable if the address in question resides in the processor's local memory. Even though multiprocessor architectures without caches are rare, we will still ask, when we consider a synchronization protocol that involves spinning, whether it permits each processor to spin on its own local memory.

On an SMP or NUMA architecture with caches, spinning consumes significantly fewer resources. The first time the processor reads the address, it takes a cache miss, and loads the contents of that address into a cache line. Thereafter, as long as those data remain unchanged, the processor simply rereads from its own cache, consuming no interconnect bandwidth, a process known as *local spinning*. When the cache state changes, the processor takes a single cache miss, observes that the data have changed, and stops spinning.

B.6 Cache-conscious programming, or the puzzle solved

We now know enough to explain why the `TTASLock` examined in Appendix B.1 outperforms the `TASLock`. Each time the `TASLock` applies `getAndSet`(*true*) to the lock, it sends a message on the interconnect causing a substantial amount of traffic. In an SMP architecture, the resulting traffic may be enough to saturate the interconnect, delaying all threads, including a thread trying to release the lock, or even threads not contending for the lock. By contrast, while the lock is busy, the `TTASLock` spins, reading a locally cached copy of the lock, and producing no interconnect traffic, explaining its improved performance.

The TTASLock is still far from ideal. When the lock is released, all its cached copies are invalidated, and all waiting threads call getAndSet(*true*), resulting in a burst of traffic, smaller than that of the TASLock, but nevertheless significant.

We further discuss the interactions of caches with locking in Chapter 7. Here we consider some simple ways to structure data to avoid false sharing.

- Objects or fields that are accessed independently should be aligned and padded so that they end up on different cache lines.
- Keep read-only data separate from data that are modified frequently. For example, consider a list whose structure is constant, but whose elements' value fields change frequently. To ensure that modifications do not slow down list traversals, one could align and pad the value fields so that each one fills up a cache line.
- When possible, split an object into thread-local pieces. For example, a counter used for statistics could be split into an array of counters, one per thread, each one residing on a different cache line. Splitting the counter allows each thread to update its own replica, avoiding the invalidation traffic that would be incurred by having a single shared counter.
- If a lock protects data that is frequently modified, then keep the lock and the data on distinct cache lines, so that threads trying to acquire the lock do not interfere with the lock-holder's access to the data.
- If a lock protects data that are frequently uncontended, then try to keep the lock and the data on the same cache lines, so that acquiring the lock will also load some of the data into the cache.

B.7 Multicore and multithreaded architectures

In a *multicore* architecture, as in Fig. B.6, multiple processors are placed on the same chip. Each processor on that chip typically has its own L1 cache, but they share a

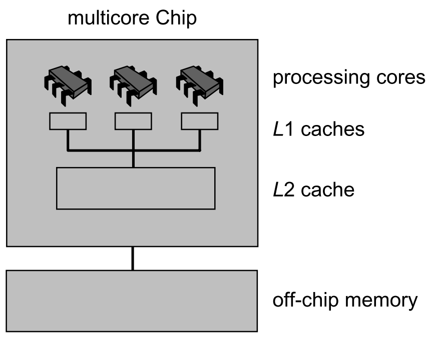

FIGURE B.6

A multicore SMP architecture. The *L*2 cache is on chip and shared by all processors while the memory is off-chip.

common L2 cache. Processors can communicate efficiently through the shared L2 cache, avoiding the need to go through memory, and to invoke the cumbersome cache-coherence protocol.

In a *multithreaded* architecture, a single processor may execute two or more threads at once. Many modern processors have substantial internal parallelism. They can execute instructions out of order, or in parallel (e.g., keeping both fixed and floating-point units busy), or even execute instructions speculatively before branches or data have been computed. To keep hardware units busy, multithreaded processors can mix instructions from multiple streams.

Modern processor architectures combine multicore with multithreading, where multiple individually multithreaded cores may reside on the same chip. The context switches on some multicore chips are inexpensive and are performed at a very fine granularity, essentially context switching on every instruction. Thus, multithreading serves to hide the high latency of accessing memory: Whenever a thread accesses memory, the processor allows another thread to execute.

B.7.1 Relaxed memory consistency

When a processor writes a value to memory, that value is kept in the cache and marked as *dirty*, meaning that it must eventually be written back to main memory. On most modern processors, write requests are not applied to memory when they are issued. Rather, they are collected in a hardware queue called a *write buffer* (or *store buffer*), and applied to memory together at a later time. A write buffer provides two benefits. First, it is often more efficient to issue several requests together, a phenomenon called *batching*. Second, if a thread writes to an address more than once, earlier requests can be discarded, saving a trip to memory, a phenomenon called *write absorption*.

The use of write buffers has a very important consequence: The order in which reads and writes are issued to memory is not necessarily the order in which they occur in the program. For example, recall the flag principle of Chapter 1, which was crucial to the correctness of mutual exclusion: If two processors each first write their own flag and then read the other's flag location, then one of them will see the other's newly written flag value. With write buffers, this is no longer true: both threads may write, each in its respective write buffer, but these writes might not be applied to the shared memory until after each processor reads the other's flag location in memory. Thus, neither reads the other's flag.

Compilers make matters even worse. They are good at optimizing performance on single-processor architectures. Often, this optimization involves *reordering* a thread's reads and writes to memory. Such reordering is invisible for single-threaded programs, but it can have unexpected consequences for multithreaded programs in which threads may observe the order in which writes occur. For example, if one thread fills a buffer with data and then sets an indicator to mark the buffer as full, then concurrent threads may see the indicator set before they see the new data, causing them to read stale values. The erroneous *double-checked locking* pattern described in Appendix A.3 is an example of a pitfall produced by unintuitive aspects of the Java memory model.

Different architectures provide different guarantees about the extent to which memory reads and writes can be reordered. As a rule, it is better not to rely on such guarantees, and instead to use more expensive techniques, described in the following paragraph, to prevent such reordering.

Every architecture provides a *memory barrier* instruction (sometimes called a *fence*), which forces writes to take place in the order they are issued, but at a price. A memory barrier flushes the write buffer, ensuring that all writes issued before the barrier become visible to the processor that issued the barrier. Memory barriers are often inserted automatically by atomic read–modify–write operations such as `getAndSet()`, or by standard concurrency libraries. Thus, explicit use of memory barriers is needed only when processors perform read–write instructions on shared variables outside of critical sections.

On one hand, memory barriers are expensive (hundreds of cycles, maybe more), and should be used only when necessary. On the other hand, synchronization bugs can be very difficult to track down, so memory barriers should be used liberally, rather than relying on complex platform-specific guarantees about limits to memory instruction reordering.

The Java language itself allows reads and writes to object fields to be reordered if they occur outside **`synchronized`** methods or blocks. Java provides a **`volatile`** keyword that ensures that reads and writes to a **`volatile`** object field that occur outside **`synchronized`** blocks or methods are not reordered. (The `atomic` template provides similar guarantees for C++.) Using this keyword can be expensive, so it should be used only when necessary. We note that in principle, one could use volatile fields to make double-checked locking work correctly, but there would not be much point, since accessing volatile variables requires synchronization anyway.

Here ends our primer on multiprocessor hardware. We now continue to discuss these architectural concepts in the context of specific data structures and algorithms. A pattern will emerge: The performance of multiprocessor programs is highly dependent on synergy with the underlying hardware.

B.8 Hardware synchronization instructions

As discussed in Chapter 5, any modern multiprocessor architecture must support powerful synchronization primitives to be universal, that is, to provide concurrent computation's equivalent of a universal Turing machine. It is therefore not surprising that implementations of Java and C++ rely on such specialized hardware instructions (also called hardware primitives) in implementing synchronization, from spin locks and monitors to the most complex lock-free data structures.

Modern architectures typically provide one of two kinds of universal synchronization primitives. The *compare-and-swap* (CAS) instruction is supported in architectures by AMD, Intel, and Oracle. It takes three arguments: an address a in memory, an *expected* value e, and an *update* value v, and returns a Boolean. It *atomically* executes the following: If the memory at address a contains the expected value e, write