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

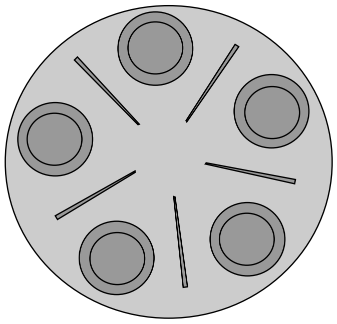

FIGURE 1.5

Traditional dining table arrangement according to Dijkstra.