CHAPTER 6

SMP Parallel Programming

in which we take a closer look at the principal constructs for SMP parallel programming; we learn how to declare shared and non-shared variables; and we gain some insight into how a Parallel Java program achieves parallelism on an SMP parallel computer

CHAPTER 6 SMP Parallel Programming

6.1 Parallel Team

A sequential program to do some computation, as embodied in the main program class's `main()` method, looks like this:

Initial setup

Execute the computation

Clean up

To change this to an SMP parallel program, we change the middle step to use a **parallel team**:

Initial setup

Create a parallel team

Have the parallel team execute the computation

Clean up

In Parallel Java, you get a parallel team by creating an instance of class ParallelTeam. The parallel team comprises a certain number of **threads**, which will be the ones to carry out the computation in parallel. You have three choices for specifying the number of threads in the team. You can specify the number of threads at compile time as the constructor argument.

```
new ParallelTeam(4); // Create a team of four threads
```

You can specify the number of threads at run time by using the no-argument constructor.

```
new ParallelTeam();
```

In this case, you specify the number of threads by defining the `"pj.nt"` Java system property when you run the program. You can do this by including the `"-Dpj.nt"` flag on the Java command line. For example,

```
$ java -Dpj.nt=4 . . .
```

specifies four threads. Finally, if you use the no-argument constructor and you do not define the `"pj.nt"` property, Parallel Java chooses the number of parallel team threads automatically to be the same as the number of processors on the computer where the program is running. (Parallel Java discovers the number of processors by calling the `Runtime.availableProcessors()` method, which is part of the standard Java platform.)

Normally, you will let Parallel Java choose the number of threads automatically at run time, so as to get the maximum possible degree of parallelism. Sometimes you will specify the number of threads explicitly at run time—for example, when you are measuring the parallel program's performance as a function of the number of processors. Certain parallel programming patterns, such as "overlapping" (which we will study in Chapter 18), always use the same number of threads, and when utilizing these patterns, you specify the number of threads at compile time.

Once a parallel team has been created, *K* threads are in existence inside the parallel team object (Figure 6.1). These are in addition to the Java program's main thread, the thread that is executing the main program class's `main()` method and that created the parallel team object. Each parallel team thread has a **thread index** from 0 through *K*–1. Each thread is poised to execute the program's computation. However, these threads are hidden, and you do not manipulate them directly in your program. To get the threads to execute the program's computation, you use another Parallel Java class, namely class ParallelRegion.

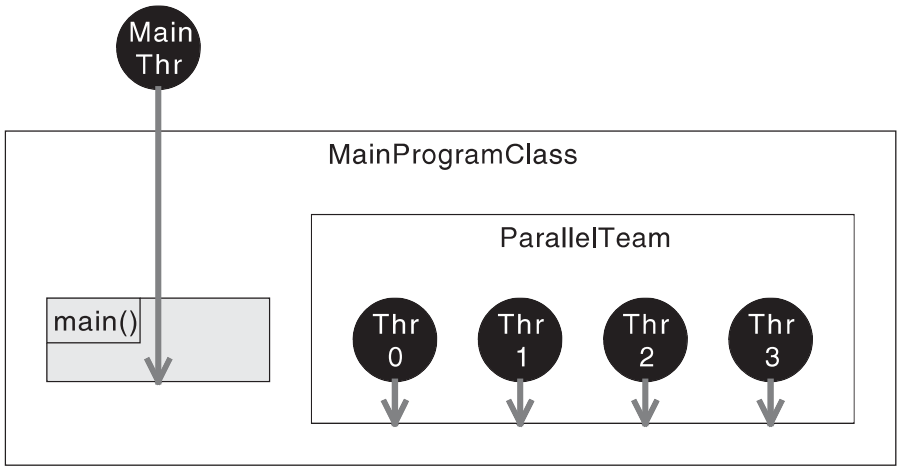

Figure 6.1 A parallel team witk *K* = 4 threads

6.2 Parallel Region

Whereas a parallel team contains the *threads* that will carry out the program's computation, a **parallel region** contains the actual *code* for the computation. Class ParallelRegion is an abstract base class with three principal methods—`start()`, `run()`, and `finish()`—that you can override. To write the parallel portion of your program, define a subclass of class ParallelRegion and put the code for the parallel computation in the subclass's `start()`, `run()`, and `finish()` methods. To make the parallel team execute

the code in the parallel region, create an instance of your ParallelRegion subclass and pass that instance to the parallel team's `execute()` method. A convenient coding idiom is to use an **anonymous inner class**.

```
new ParallelTeam().execute (new ParallelRegion()
  {
  public void start()
     {
     // Initialization code
     }
  public void run()
     {
     // Parallel computation code
     }
  public void finish()
     {
     // Finalization code
     }
  });
```

This code creates a new parallel team object, defines the ParallelRegion subclass as an anonymous inner class, creates an instance of the parallel region subclass, and passes the instance to the parallel team's `execute()` method.

Here's what happens inside the parallel team (Figure 6.2). Each parallel team thread is initially blocked and will not commence execution until signaled. The main thread, executing the program's `main()` method, calls the parallel team's `execute()` method. Inside the `execute()` method, the main thread calls the parallel region's `start()` method. When the `start()` method returns, the main thread signals each team thread to commence execution. The main thread then blocks.

All the team threads now proceed to call the parallel region's `run()` method. Here is where the parallel execution of the parallel region code happens. When the `run()` method returns, each team thread signals the main thread. At this point, the team threads block again, waiting for the team to execute another parallel region.

Once the main thread has been signaled by each of the team threads, the main thread resumes execution and calls the parallel region's `finish()` method. When the `finish()` method returns, the parallel team's `execute()` method also returns, and the main thread continues executing whatever comes afterward in the program's `main()` method.

It's important to emphasize that you don't have to do the preceding steps yourself. Parallel Java does it all for you. You merely have to write the code you want executed in the parallel region's `start()`, `run()`, and `finish()` methods. However, to design your parallel program properly, you have to understand what's going on "under the hood."

Focusing on the parallel region's methods in Figure 6.2, the sequence of execution is the following:

- The `start()` method is executed by a single thread.
- The `run()` method is executed by *K* threads simultaneously.
- The `finish()` method is executed by a single thread.

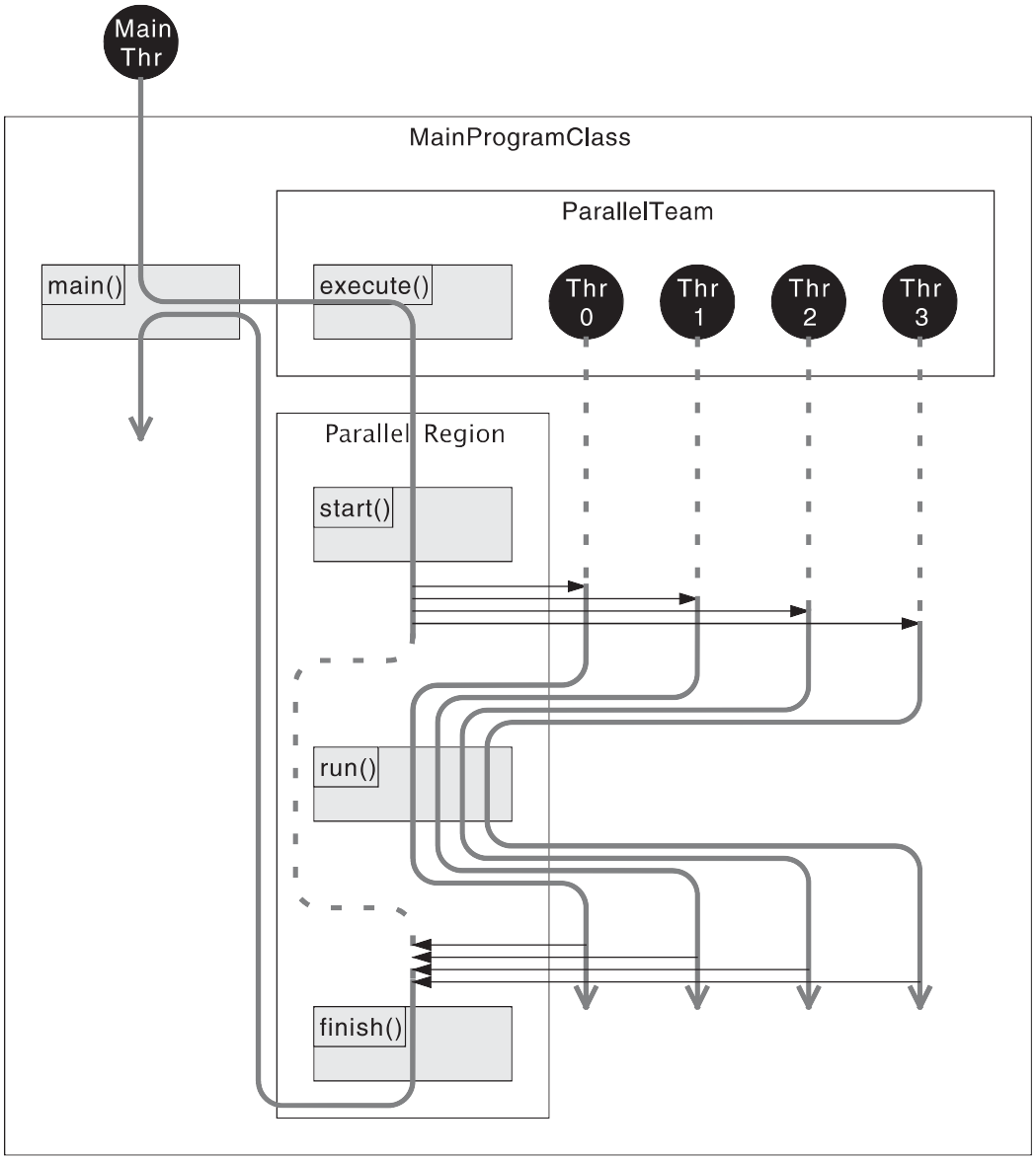

Figure 6.2 A parallel team executing a parallel region

Thus, in the `start()` method, put any initialization code that must be executed in a single thread before the parallel computation starts. In the `run()` method, put the parallel computation code itself. In the `finish()` method, put any finalization code that must be executed in a single thread after the parallel computation finishes. If no initialization or finalization code is necessary, simply omit the `start()` or `finish()` method or both.

As already mentioned, when running a program with a parallel team and a parallel region on an SMP parallel computer, the JVM and the operating system are responsible for scheduling each thread to execute on a different processor. To see a parallel speedup, the parallel region's `run()` method must divide the computation among the K team threads—that is, among the K processors. With all processors executing simultaneously and each processor doing $1/K$ of the total work, the program should experience a speedup.

6.3 Parallel For Loop

Often, a program's computation consists of some number *N* of loop iterations. To divide the *N* loop iterations among the *K* threads (processors), you use yet another Parallel Java class, IntegerForLoop, which provides a **parallel for loop**. Class IntegerForLoop is an abstract base class with three principal methods—`start()`, `run()`, and `finish()`—that you can override. To write the parallel for loop, define a subclass of class IntegerForLoop and put the loop code in the subclass's `start()`, `run()`, and `finish()` methods. Then, in the parallel region's `run()` method, call the parallel region's `execute()` method, passing in the first loop index (inclusive), the last loop index (inclusive), and the IntegerForLoop subclass instance. Again, a convenient coding idiom is to use an anonymous inner class. Here is a parallel for loop with the index going from 0 to 99, inclusive.

```
new ParallelTeam().execute (new ParallelRegion()
  {
  public void run()
     {
     execute (0, 99, new IntegerForLoop()
        {
        public void start()
           {
           // Per-thread pre-loop initialization code
           }
        public void run (int first, int last)
           {
           // Loop code
           }
        public void finish()
           {
           // Per-thread post-loop finalization code
           }
        });
     }
  });
```

Here's what happens inside the parallel team (Figure 6.3). The parallel team threads are executing the parallel region's `run()` method simultaneously. Each team thread executes the following statement:

```
execute (0, 99, new IntegerForLoop()...);
```

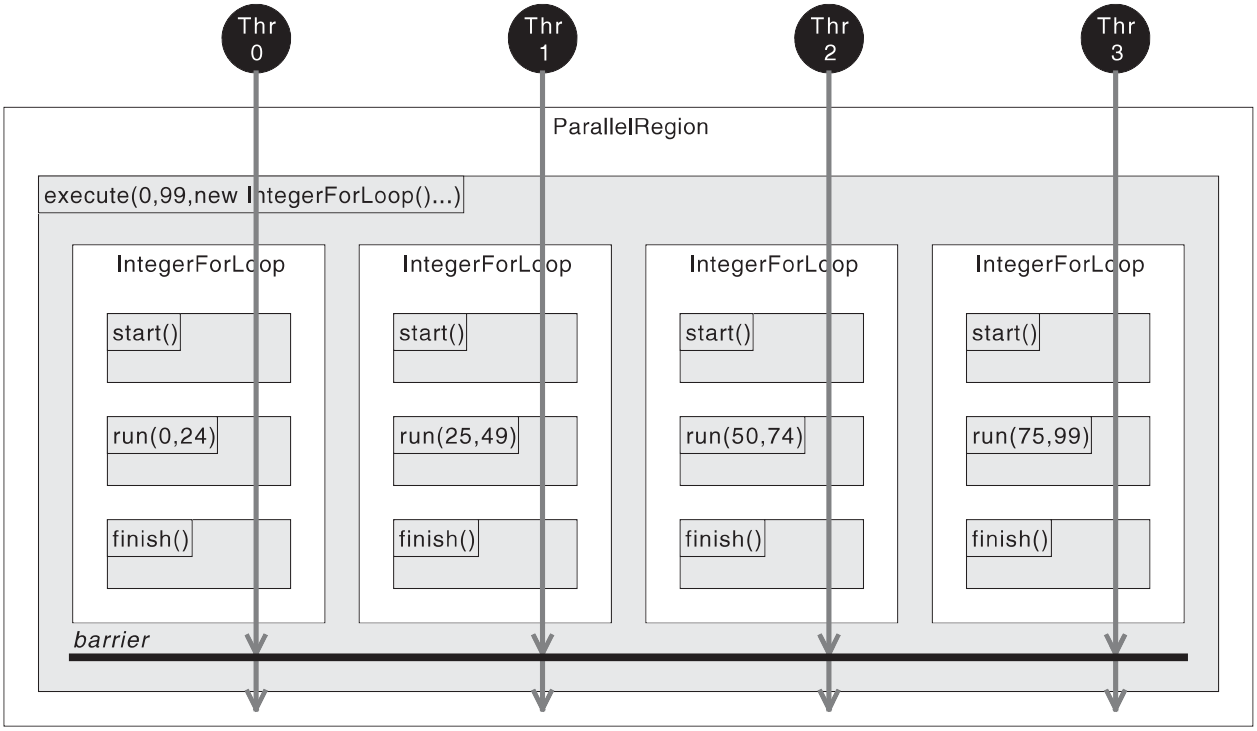

Figure 6.3 A parallel team executing a parallel for loop

Each thread, therefore, first creates its own new instance of the IntegerForLoop subclass. Each thread then calls the parallel region's `execute()` method, passing in the loop index lower bound (0), the loop index upper bound (99), and the thread's own IntegerForLoop object. (All threads must pass in the *same* loop index lower and upper bounds, and these must be the bounds for the *whole* loop.) The parallel region's `execute()` method partitions the complete index range, 0–99 in this example, into *K* equal-sized **subranges**, or **chunks**, namely 0–24, 25–49, 50–74, and 75–99. Each thread now proceeds to call its own IntegerForLoop object's `start()`, `run()`, and `finish()` methods in sequence. However, each thread passes a different chunk to the `run()` method. Thread 0 passes the arguments first = 0 and last = 24, thread 1 passes the arguments 25 and 49, thread 2 passes the arguments 50 and 74, and thread 3 passes the arguments 75 and 99. After completing this sequence of execution through its IntegerForLoop object, each thread waits at a **barrier**. A barrier is a thread synchronization mechanism that ensures that none of the threads will proceed past the barrier until all the threads have arrived at the barrier. When the last thread finishes its portion of the parallel for loop and arrives at the barrier, like the final horse arriving at the starting gate for the Kentucky Derby, the barrier opens. Each thread resumes execution, returns from the parallel region's `execute()` method, and continues executing the code that comes after the parallel for loop in the parallel region's `run()` method.

As with the parallel region, it's important to emphasize that you don't have to do the preceding steps. Parallel Java does it all for you. You merely have to write the code you want executed in the parallel for loop's `start()`, `run()`, and `finish()` methods.

The purpose of the parallel for loop's `start()` and `finish()` methods is to do any necessary initialization within the parallel for loop before beginning the actual loop iterations, and to do any necessary finalization after finishing the loop iterations. If no initialization or finalization code is necessary, simply omit the `start()` or `finish()` method or both.

The parallel for loop's `run()` method's job is to execute the loop iterations for the chunk whose first and last loop indexes, inclusive, are passed in as arguments. Thus, the `run()` method typically looks like this.

```
public void run (int first, int last)
  {
  for (int i = first; i <= last; ++ i)
     {
     // Code for loop iteration i
     }
  }
```

Note that your code does not decide which chunk a particular call of the `run()` method will perform. The parallel region decides that. The `run()` method must do exactly the loop iterations specified by the `first` and `last` arguments, no more, no less.

Figure 6.3 shows where the parallel for loop's speedup comes from. Instead of executing 100 loop iterations in sequence as a regular program would do, the parallel program executes four chunks of 25 iterations in parallel, each chunk being executed by a different thread. Each thread (processor) does $1/K$ of the total work, resulting in a speedup.

Parallel Java also has classes for doing a parallel loop with an index of type `long`, and for doing a parallel loop over a collection of objects instead of a range of indexes. For further information, refer to the Parallel Java documentation.

6.4 Variables

Having looked at where to put the *code* for a Parallel Java program, let us turn our attention to where to put the *variable declarations* for a Parallel Java program.

Following the aforementioned coding idioms gives rise to a nested class structure (Figure 6.4). The parallel for loop class is nested inside the parallel region class, which in turn is nested inside the main program class.

```
public class MainProgramClass
  {
  // Shared variable declarations
  static int a;

  public static void main
     (String[] args)
     throws Exception
     {
     // Main program local variable declarations
     int b;
```

```
      new ParallelTeam().execute (new ParallelRegion()
         {
         public void run()
             {
             execute (0, 99, new IntegerForLoop()
                {
                // Per-thread variable declarations
                int c;

                public void run (int first, int last)
                   {
                   // Loop local variable declarations
                   int d;
                   for (int i = first; i <= last; ++ i)
                      {
                      // Code for loop iteration i
                      }
                   }
                });
             }
         });
      }
   }
```

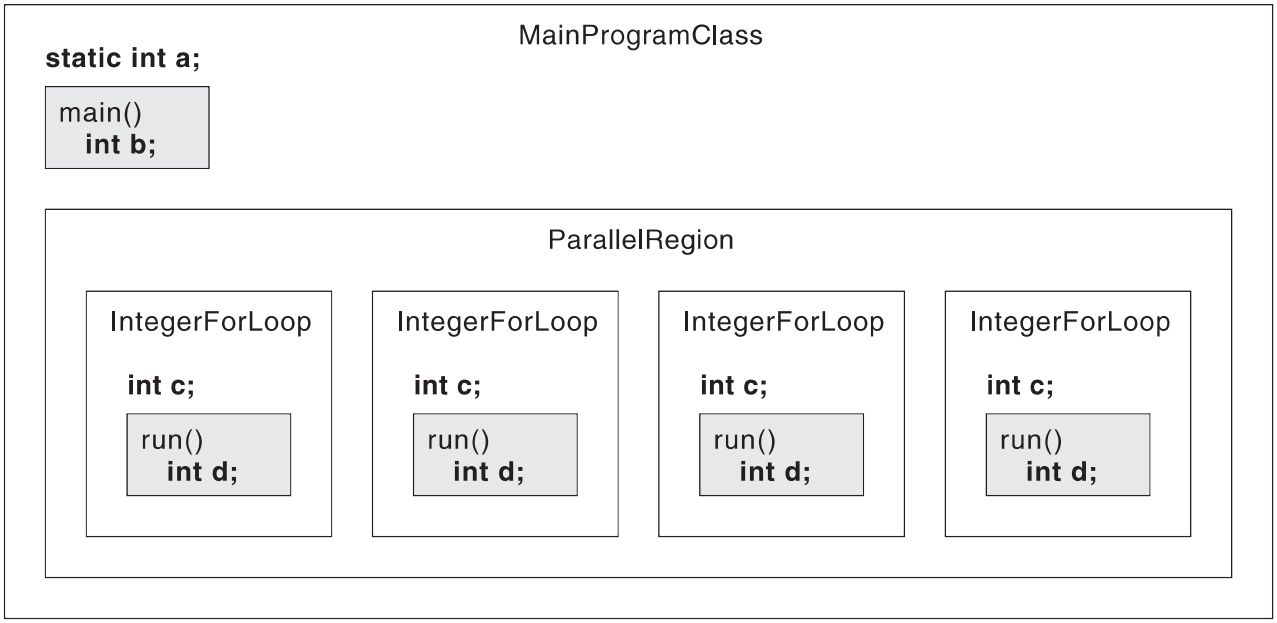

Figure 6.4 Variable declarations in a Parallel Java program

Variables can be declared either as fields of these classes, or as local variables of the classes' methods. In practice, there are four categories of variables in a Parallel Java program; they differ in the places they are declared and in the ways they are accessed.

Shared variables (such as `a`) are declared as static fields of the main program class. Because they are *static* fields, there is only one instance of each shared variable. Furthermore, each shared variable can be accessed by code in the static `main()` method, by code in other static methods of the main program class, and by code in the parallel region and parallel for loop subclasses. Thus, the main program thread and all the parallel team threads can access every shared variable. These variables are called "shared" variables to emphasize that all the threads access the *same* instance of each variable. If one thread changes the value of a shared variable, all the other threads will see that new value.

Per-thread variables (such as `c`) are declared as instance fields of the parallel for loop subclass. Such variables can be accessed by code anywhere in the parallel for loop subclass. Because each thread creates its own instance of the parallel for loop subclass, each thread gets its own separate instances of the per-thread variables. If one thread changes the value of a per-thread variable, this will not affect the value of the corresponding per-thread variable in any other thread; per-thread variables are *not* shared. The parallel for loop's `start()` and `finish()` methods can be used to initialize and finalize these per-thread variables.

Loop local variables (such as `d`) are declared as local variables of the parallel for loop subclass's `run()` method. The loop control variable (such as `i`) is also a loop local variable. Loop local variables can be accessed only by code in the parallel for loop's `run()` method, and each thread gets its own separate instance of each loop local variable.

Main program local variables (such as `b`) are declared as local variables of the static `main()` method. Such variables are used only by the main thread executing the `main()` method. (Code inside the parallel region or parallel for loop cannot access main program local variables.)

An SMP parallel program is organized around a data structure or data structures located in **shared memory**, that is, in memory accessed by all the processors (threads) in the SMP parallel computer (Figure 6.5). In a result parallel program, the shared data structure may contain all the program's results. In an agenda parallel program, the shared data structure may contain the program's agenda items and their results. In a specialist parallel program, the shared data structure may contain certain tasks' outputs, which become other tasks' inputs. To get a parallel speedup, *all* the threads must access the *same* data structure, each thread working with a different piece of the data structure simultaneously. In a Parallel Java program, a data structure is made to reside in shared memory by declaring the data structure as a shared variable. The JVM then ensures that all the threads access the same instance of the variable.

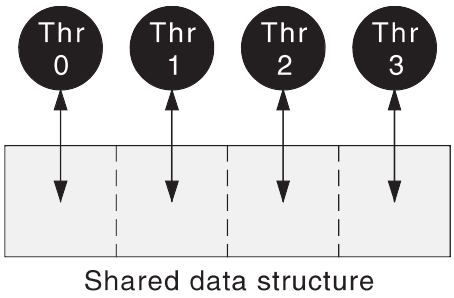

Figure 6.5 Shared data structure, each thread accessing its own portion

If a variable is shared by multiple threads, however, we must make sure that the threads do not **conflict** with each other when they access the shared variable. A thread can do certain operations on a variable:

- A thread can **read** a variable; that is, the thread can examine the variable's state without changing its state.
- A thread can **write** a variable; that is, the thread can change the variable's state.
- A thread can **update** a variable; that is, the thread can read the variable's state, compute a new state based on the old state, and write the new state back into the variable.

A conflict can arise when two or more threads do certain operations on a variable at the same time:

- A **read-write conflict** can arise if one thread reads a variable at the same time as another thread writes or updates the variable; the reading thread may read an inconsistent state where part of the state is the old state and part of the state is the new state.
- A **write-write conflict** can arise if two threads write or update a variable at the same time; one thread's writes may wipe out some of the other thread's writes, again leading to an inconsistent state.
- However, there is no conflict if two threads read a variable at the same time.

If multiple threads cannot conflict when they access a shared variable, then we don't have to do anything special. But if multiple threads can conflict when they access a shared variable, we must **synchronize** the threads to eliminate the potential conflict. Parallel Java has several constructs for thread synchronization that we will study later. As we will see, thread synchronization adds overhead to the parallel program and can significantly increase the program's running time. Because the goal of parallel programming is to *reduce* the running time, we must carefully analyze each shared variable for thread conflicts and introduce thread synchronization only where it is absolutely needed.

A key aspect of designing a Parallel Java program, then, is to decide where in the program to declare each variable, depending on whether the variable does or does not need to be shared. A second key aspect is to decide whether and how to synchronize the multiple threads accessing each shared variable. This will be a recurring theme as we study SMP parallel programming in the chapters ahead.

With this introduction to Parallel Java's constructs for SMP parallel programming, we are ready to convert the sequential AES key search program from Chapter 5 to an SMP parallel program in Chapter 7.

6.5 For Further Information

On the concepts of multithreading and thread synchronization, see any operating systems textbook, such as:

- A. Silberschatz, P. Galvin, and G. Gagne. *Operating System Concepts, 8th Edition*. John Wiley & Sons, 2009.

- A. Tanenbaum. *Modern Operating Systems, 3rd Edition*. Prentice-Hall, 2007.
- W. Stallings. *Operating Systems: Internals and Design Principles, 5th Edition*. Prentice-Hall, 2004.
- G. Nutt. *Operating Systems*, 3rd Edition. Addison-Wesley, 2003.

On the concepts of multithreaded programming and thread synchronization in Java:

- B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice*. Addison-Wesley, 2006.
- D. Lea. *Concurrent Programming in Java: Design Principles and Patterns, 3rd Edition*. Addison-Wesley, 2006.
- Java Threads Tutorial. http://java.sun.com/docs/books/tutorial/essential/threads/ index.html