CHAPTER 4

A First Parallel Program

in which we build a simple sequential program; we convert it to a program for an SMP parallel computer; we see how long it takes to run each version; and we get some insight into how parallel programs execute

CHAPTER 4 A First Parallel Program

4.1 Sequential Program

To demonstrate a program that can benefit from running on a parallel computer, let's invent a simple computation that will take a long time. Here is a Java subroutine that decides whether a number x is prime using the **trial division** algorithm. The subroutine tries to divide x by 2 and by every odd number p from 3 up to the square root of x. If any remainder is 0, then p is a factor of x and x is not prime; otherwise x is prime. While trial division is by no means the fastest way to test primality, it suffices for this demonstration program.

```
private static boolean isPrime
   (long x)
   {
   if (x % 2 == 0) return false;
   long p = 3;
   long psqr = p*p;
   while (psqr <= x)
      {
      if (x % p == 0) return false;
      p += 2;
      psqr = p*p;
      }
   return true;
   }
```

Here is a main program that uses a loop to call the subroutine with the values of x specified on the command line.

```
static int n;
static long[] x;

public static void main
   (String[] args)
   throws Exception
   {
   n = args.length;
```

```
    x = new long [n];
    for (int i = 0; i < n; ++ i)
       {
       x[i] = Long.parseLong (args[i]);
       }
    for (int i = 0; i < n; ++ i)
       {
       isPrime (x[i]);
       }
    }
```

When we run the primality testing program, we want to know the time when each subroutine call starts and the time when each subroutine call finishes, relative to the time the program started. This tells us how long it took to run the subroutine. To measure these times, we use Java's `System.currentTimeMillis()` method, which returns the wall clock time in milliseconds (msec). We record each instant in a variable, and postpone printing the results, so as to disturb the timing as little as possible while the program is running. It can take several msec to call `println()`, and we don't want to include that time in our measurements.

```
static int n;
static long[] x;
static long t1, t2[], t3[];

public static void main
   (String[] args)
   throws Exception
   {
   t1 = System.currentTimeMillis();
   n = args.length;
   x = new long [n];
   for (int i = 0; i < n; ++ i)
      {
      x[i] = Long.parseLong (args[i]);
      }
   t2 = new long [n];
   t3 = new long [n];
   for (int i = 0; i < n; ++ i)
      {
      t2[i] = System.currentTimeMillis();
      isPrime (x[i]);
      t3[i] = System.currentTimeMillis();
      }
   }
```

Here is the complete Java class, Program1Seq, including code to print the running time measurements.

```
public class Program1Seq
    {
    static int n;
    static long[] x;
    static long t1, t2[], t3[];

 public static void main
    (String[] args)
    throws Exception
    {
    t1 = System.currentTimeMillis();
    n = args.length;
    x = new long [n];
    for (int i = 0; i < n; ++ i)
       {
       x[i] = Long.parseLong (args[i]);
       }
    t2 = new long [n];
    t3 = new long [n];
    for (int i = 0; i < n; ++ i)
       {
       t2[i] = System.currentTimeMillis();
       isPrime (x[i]);
       t3[i] = System.currentTimeMillis();
       }
    for (int i = 0; i < n; ++ i)
       {
       System.out.println
          ("i = "+i+" call start = "+(t2[i]-t1)+" msec");
       System.out.println
          ("i = "+i+" call finish = "+(t3[i]-t1)+" msec");
       }
    }

 private static boolean isPrime
    (long x)
    {
    if (x % 2 == 0) return false;
    long p = 3;
    long psqr = p*p;
    while (psqr <= x)
```

```
        {
        if (x % p == 0) return false;
        p += 2;
        psqr = p*p;
        }
    return true;
    }
}
```

4.2 Running the Sequential Program

When run on an SMP parallel computer, Program1Seq prints the following. The parallel computer has four processors, each a 450 MHz Sun Microsystems UltraSPARC-II CPU, and 2 GB of shared main memory. (Running the program on a different computer would, in general, yield different results.) All four arguments happen to be prime numbers.

```
$ java Program1Seq 1000000000000037 1000000000000091 \
  1000000000000159 1000000000000187
i = 0 call start = 1 msec
i = 0 call finish = 3842 msec
i = 1 call start = 3842 msec
i = 1 call finish = 7663 msec
i = 2 call start = 7663 msec
i = 2 call finish = 11502 msec
i = 3 call start = 11502 msec
i = 3 call finish = 15342 msec
```

Plotting each subroutine call's start and finish on a timeline reveals how the program executes (Figure 4.1). The program executes each subroutine call in its entirety before going on to the next subroutine call. Because the program's statements are executed in sequence with no overlap in time, we call it a **sequential program**. There is no parallelism, even when running on a parallel computer.

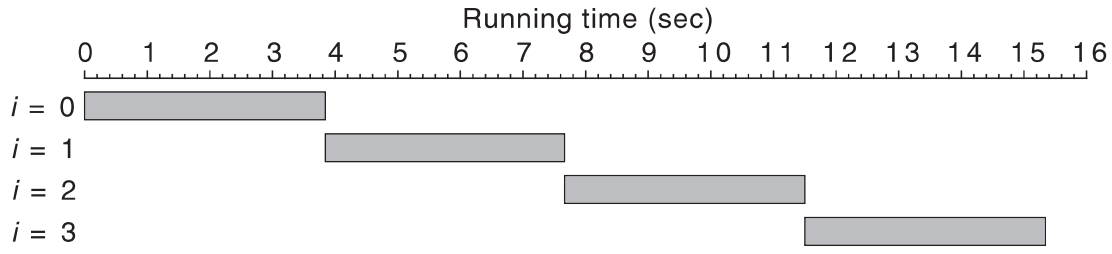

Figure 4.1 Program1Seq execution timeline, SMP parallel computer

4.3 SMP Parallel Program

Now let's rewrite the program using Parallel Java so it will run in parallel when executed on an SMP parallel computer. In the main program, after extracting the command line arguments, we create a **parallel team** object. The constructor argument, `n`, says we want as many threads in the parallel team as there are values to test for primality.

```
public static void main
   (String[] args)
   throws Exception
   {
   n = args.length;
   x = new long [n];
   for (int i = 0; i < n; ++ i)
      {
      x[i] = Long.parseLong (args[i]);
      }
   new ParallelTeam(n);
   }
```

Each thread in the parallel team simultaneously executes the code in a **parallel region** object, declared here as an anonymous inner class. The actual parallel code goes in the parallel region's `run()` method.

```
public static void main
   (String[] args)
   throws Exception
   {
   n = args.length;
   x = new long [n];
   for (int i = 0; i < n; ++ i)
      {
      x[i] = Long.parseLong (args[i]);
      }
   new ParallelTeam(n).execute (new ParallelRegion()
      {
      public void run()
         {
         }
      });
   }
```

Rather than use a loop to execute the computations (subroutine calls) in sequence, we want the threads to execute the computations in parallel. To make this happen, we put the code for one computation in the parallel region's `run()` method. However, we want each computation to use a different x value. To make this happen, we set `i` to the index of the calling thread within the parallel team (0 through 3, as returned by the parallel region's `getThreadIndex()` method).

```
 public static void main
    (String[] args)
    throws Exception
    {
    n = args.length;
    x = new long [n];
    for (int i = 0; i < n; ++ i)
       {
       x[i] = Long.parseLong (args[i]);
       }
    new ParallelTeam(n).execute (new ParallelRegion()
       {
       public void run()
          {
          int i = getThreadIndex();
          isPrime (x[i]);
          }
       });
    }
```

Here is the complete Java class, Program1Smp, including code to record the running time measurements and print them after the parallel region has finished executing.

```
import edu.rit.pj.ParallelRegion;
import edu.rit.pj.ParallelTeam;
public class Program1Smp
    {
    static int n;
    static long[] x;
    static long t1, t2[], t3[];

 public static void main
    (String[] args)
    throws Exception
```

```
      {
      t1 = System.currentTimeMillis();
      n = args.length;
      x = new long [n];
      for (int i = 0; i < n; ++ i)
         {

         x[i] = Long.parseLong (args[i]);
         }
      t2 = new long [n];
      t3 = new long [n];
      new ParallelTeam(n).execute (new ParallelRegion()
         {
         public void run()
            {
            int i = getThreadIndex();
            t2[i] = System.currentTimeMillis();
            isPrime (x[i]);
            t3[i] = System.currentTimeMillis();
            }
         });
      for (int i = 0; i < n; ++ i)
         {
         System.out.println
            ("i = "+i+" call start = "+(t2[i]-t1)+" msec");
         System.out.println
            ("i = "+i+" call finish = "+(t3[i]-t1)+" msec");
         }
      }

   private static boolean isPrime
      (long x)
      {
      if (x % 2 == 0) return false;
      long p = 3;
      long psqr = p*p;
      while (psqr <= x)
         {
         if (x % p == 0) return false;
         p += 2;
         psqr = p*p;
         }
      return true;
      }
   }
```

Here's how the program works (Figure 4.2). The main program begins with one thread, the "main thread," executing the `main()` method. When the main thread creates the parallel team object, the parallel team object creates additional hidden threads; the constructor argument specifies the number of threads. These form a "team" of threads for executing code in parallel. When the main thread calls the parallel region's `execute()` method, the main thread suspends execution and the parallel team threads take over. All the team threads call the parallel region's `run()` method simultaneously, each thread retrieves a value for *x*, and each thread calls `isPrime()`. Thus, the `isPrime()` subroutine calls happen at the same time, and each subroutine call is performed by a different thread with a different argument. When all the subroutine calls have finished executing, the main thread resumes executing statements after the parallel region and prints the timing measurements.

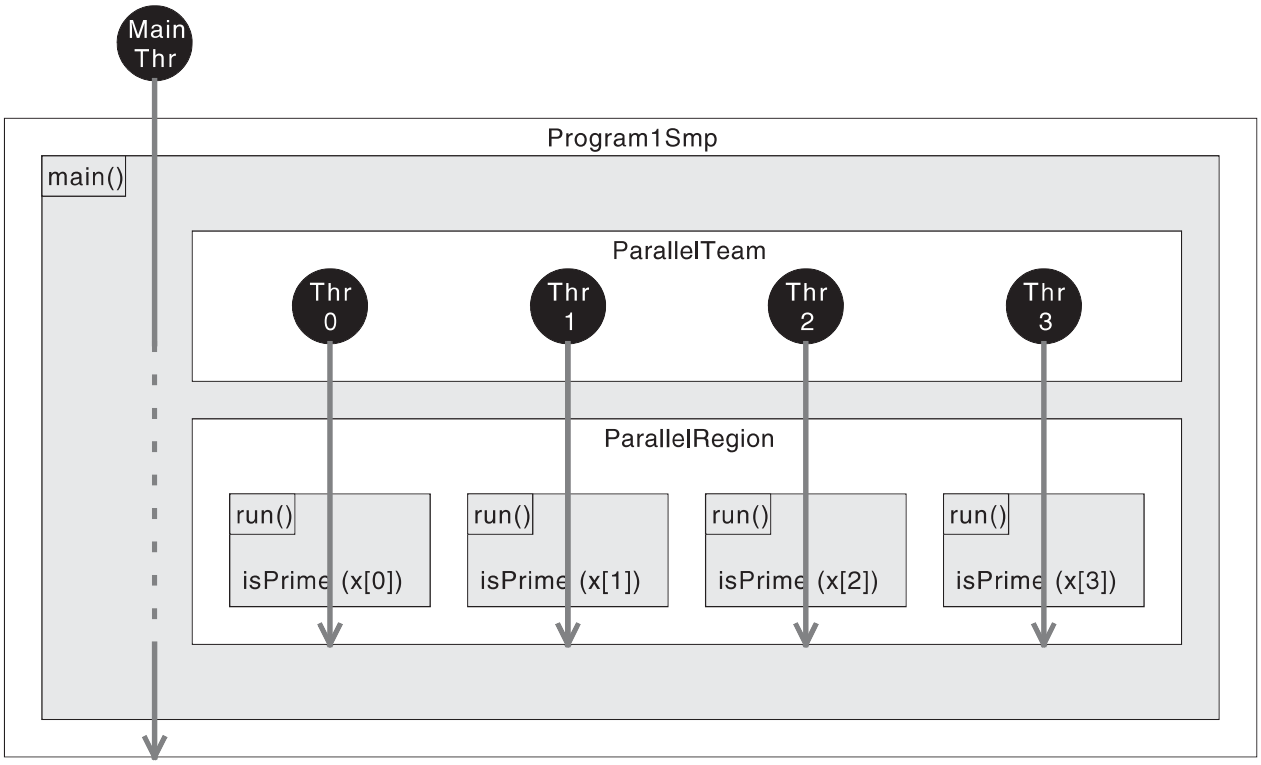

Figure 4.2 Program1Smp operation

When running such a thread-based program on an SMP parallel computer, the Java Virtual Machine (JVM) and the operating system are responsible for scheduling each thread to execute on a different processor. Thus, the computations done by each thread—in this case, the different subroutine calls—are executed in parallel on different processors, resulting in a speedup with respect to the sequential program.

The parallel program illustrates a central theme of parallel program design: *Repetition does not necessarily imply sequencing.* The sequential program used a loop to get *n* repetitions of a subroutine call. As a side effect, the loop did the repetitions *in sequence.* However, for this program there is no need to do the repetitions in sequence. We wrote the original program with a loop because Java, like many programming languages, only has constructs for expressing a *sequence* of repetitions (a loop). So accustomed are we to this feature that whenever we are confronted with a repeated calculation, we automatically think "loop." However, a loop is not the only way to do a repeated calculation. Provided the

repetitions do not have to be done in sequence, another way to do a repeated calculation is to run several copies of the calculation in multiple threads. A large part of the effort in learning parallel program design is breaking the habit of always using a loop to do repetitions in sequence, and forming the new habit of doing repetitions in parallel whenever possible.

4.4 Running the Parallel Program

When run on the four-processor parallel computer, Program1Smp printed the following:

```
$ java Program1Smp 1000000000000037 1000000000000091 \
  1000000000000159 1000000000000187
i = 0 call start = 125 msec
i = 0 call finish = 4076 msec
i = 1 call start = 125 msec
i = 1 call finish = 4098 msec
i = 2 call start = 125 msec
i = 2 call finish = 4082 msec
i = 3 call start = 125 msec
i = 3 call finish = 4076 msec
```

Now the timeline (Figure 4.3) shows parallelism. (Compare Figures 4.1 and 4.3 to Figure 1.2.) All the computations start at the same time, execute simultaneously, and finish at about the same time. Whereas the sequential version's running time was 15342 msec, the parallel version's running time on four processors was 4098 msec—a reduction by a factor of about four, as expected.

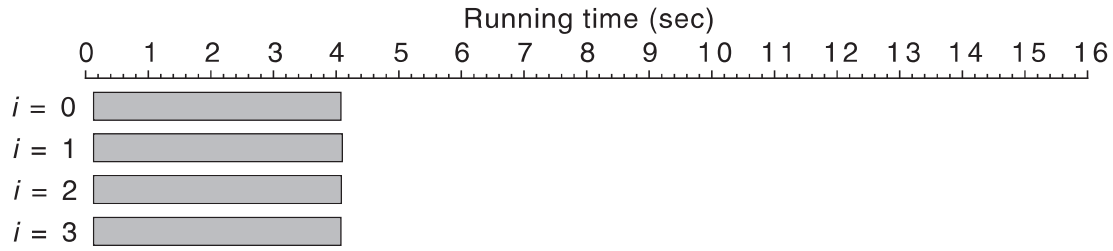

Figure 4.3 Program1Smp execution timeline, SMP parallel computer

To be precise, the speedup (the reduction factor) was 15342/4098 = 3.744. The speedup was somewhat less than 4 because of overhead in the parallel version that is not present in the sequential version. With Program1Smp, the first subroutine call didn't begin until 125 msec after the program started. During this time, the program was occupied in creating the parallel team and parallel region objects, starting up the parallel team threads, and executing the parallel region's `run()` method—work that the sequential program didn't have to do.

This illustrates another central theme of parallel program design: *Parallelism is not free.* The benefit of speedup or sizeup comes with a price of extra overhead that is not needed in a sequential program. The name of the game is to minimize this extra overhead.

4.5 Running on a Regular Computer

Although intended to run on a parallel computer, Program1Seq and Program1Smp are perfectly happy to run on a nonparallel computer. In fact, one benefit of programming in Parallel Java is that you can develop and test parallel programs on any computer, and then you can shift to a parallel computer when the program is debugged and ready for usage.

Let's look at what happens when we run these programs on a regular computer. This was a nonparallel computer with a 1.6 GHz Intel Pentium CPU and 512 MB of main memory. The sequential Program1Seq program printed the following:

```
$ java Program1Seq 1000000000000037 1000000000000091 \
  1000000000000159 1000000000000187
i = 0 call start = 0 msec
i = 0 call finish = 1594 msec
i = 1 call start = 1594 msec
i = 1 call finish = 2881 msec
i = 2 call start = 2881 msec
i = 2 call finish = 4165 msec
i = 3 call start = 4165 msec
i = 3 call finish = 5450 msec
```

The timeline (Figure 4.4) shows the typical pattern of sequential execution.

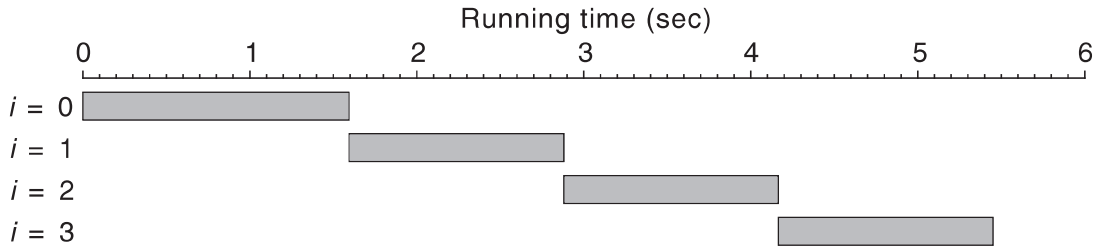

Figure 4.4 Program1Seq execution timeline, regular computer

Suppose we run the multithreaded Program1Smp program on the regular computer. Because each subroutine call will now run in a different thread, we would expect all the subroutine calls to start at roughly the same time near the beginning of the program. But because all the threads will share the same processor, and the processor will execute one thread at a time and switch to another thread every so often, we would expect the overall running time to be about the same as the sequential program. Here is what the parallel program printed.

```
$ java Program1Smp 1000000000000037 1000000000000091 \
  1000000000000159 1000000000000187
i = 0 call start = 21 msec
i = 0 call finish = 5190 msec
```

```
i = 1 call start = 71 msec
i = 1 call finish = 5053 msec
i = 2 call start = 91 msec
i = 2 call finish = 5134 msec
i = 3 call start = 14 msec
i = 3 call finish = 4981 msec
```

The timeline for this run (Figure 4.5) is about what we expected—except for one thing. The overall running time was 260 msec shorter for the four-thread parallel version than for the single-thread sequential version. We got a slight but noticeable speedup when we went from one thread to four threads on the regular computer. How can this be, when there was only one processor?

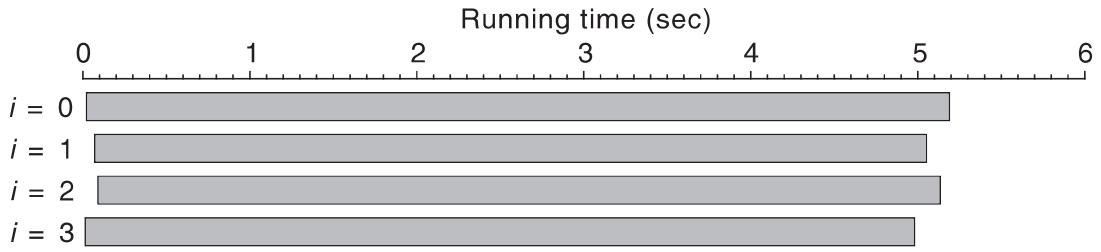

Figure 4.5 Program1Smp execution timeline, regular computer

The reason has to do with how the JVM works. A modern JVM includes a **just-in-time (JIT) compiler** that converts the Java bytecode instructions into native machine code instructions as the program runs. The JVM then executes the machine code directly instead of interpreting the Java bytecode; this greatly increases the program's execution speed. Furthermore, a modern JVM monitors which sections of bytecode are executed most frequently and compiles just those sections to machine code, leaving the remaining sections as interpreted bytecode. This avoids spending the time it would take to compile infrequently used sections of bytecode. (Sun Microsystems refers to this as a **HotSpot JVM**.) However, it takes a certain amount of execution before the JVM detects that the `isPrime()` subroutine is a hot spot and compiles it to machine code. With four threads all calling the subroutine at the same time, the JVM can detect the hot spot, and compile it to machine code, sooner in the parallel version than in the sequential version. This allows more of the parallel version's running time to be executed in the faster machine code mode, thus reducing the parallel version's running time compared to the sequential version. (To verify that this is in fact what's going on, try running both versions with the JIT compiler disabled; the parallel version then invariably takes longer than the sequential version due to the parallel version's extra overhead.) We will see further instances of how the JVM's behavior influences program performance as we study parallel programming in Java.

4.6 The Rest of the Book

Let's step back and look at what we've done. We started with a problem statement. We wrote a sequential program and a parallel program to solve the problem. The parallel program illustrated both general parallel programming techniques (in this case, achieving repetition via multiple threads) and specific Parallel

Java features (parallel team and parallel region). Then we ran the sequential and parallel programs, measured their running times, and gained some insight about parallel programming by comparing the programs' performance.

The rest of the book will be much the same—solving a series of problems that are chosen to illustrate various parallel programming techniques and studying the programs' performance measurements. In Part II, we will begin with SMP parallel programs, because those are quite similar to regular sequential programs. Then, in Part III, we will move on to cluster parallel programs, which are a bit more different from regular sequential programs due to the explicit message passing that is needed. In Part IV, we will combine techniques for SMP parallel programming and techniques for cluster parallel programming to write hybrid parallel programs. While the problems we solve in Parts II through IV will be interesting and perhaps fun, they were chosen solely for pedagogical reasons—to illustrate parallel programming techniques—and are not necessarily problems with any great significance in the real world. Finally, in Part V, we will apply the techniques we've learned to solve some *real-world* problems using parallel computing.

4.7 For Further Information

On the HotSpot JVM, and performance tuning of Java programs in general:

- Steve Wilson and Jeff Kesselman. *Java Platform Performance: Strategies and Tactics*. Addison-Wesley, 2000. Available online at: http://java.sun.com/docs/books/performance/
- Java Performance Documentation. http://java.sun.com/docs/performance/index.html