Chapter 4
Parallel Reduction

▶ Part I. Preliminaries
▼ Part II. Tightly Coupled Multicore
   Chapter 2. Parallel Loops
   Chapter 3. Parallel Loop Schedules
   Chapter 4. Parallel Reduction
   Chapter 5. Reduction Variables
   Chapter 6. Load Balancing
   Chapter 7. Overlapping
   Chapter 8. Sequential Dependencies
   Chapter 9. Strong Scaling
   Chapter 10. Weak Scaling
   Chapter 11. Exhaustive Search
   Chapter 12. Heuristic Search
▶ Part III. Loosely Coupled Cluster
▶ Part IV. GPU Acceleration
▶ Part V. Map-Reduce
Imagine a square dartboard (Figure 4.1) with sides of length 1 and with a quadrant of a circle drawn in it. Suppose we throw a large number $N$ of darts at the dartboard, and suppose $C$ of them land inside the circle quadrant. Assuming the darts land at random positions, the ratio $C/N$ should be approximately the same as the ratio of the circle quadrant’s area $= \pi/4$ to the dartboard’s area $= 1$:

$$C/N \approx \pi/4,$$ or $$\pi \approx 4C/N.$$ 

This suggests an algorithm for approximating $\pi$: Generate $N$ random $(x, y)$ points in the unit square; count how many lie at a distance of 1 or less from the origin ($x^2 + y^2 \leq 1$) yielding $C$; report $4C/N$ as the estimate for $\pi$. However, to get an accurate estimate, $N$ needs to be very large, so we want to do the computation in parallel.

As shown in Figure 4.2, a sequential program simply throws $N$ darts, counts $C$, and calculates $\pi$. On the other hand, a parallel program running on, say, four cores partitions the computation among four threads. Each thread throws $N/4$ darts and counts how many of its own darts land within the circle quadrant. The program then has to add these per-thread counts, $C_0$ through $C_3$, together to get the total count $C$, from which the program calculates $\pi$.

The process of combining multiple parallel threads’ results into one overall result is called reduction. Here the reduction operation is addition, or sum, and we refer to the reduction as a sum-reduce. (Other programs would use other reduction operations as part of the same reduction pattern.)

Listing 4.1 is the sequential $\pi$ program. It is mostly self-explanatory. On line 28 I created a pseudorandom number generator (PRNG), an instance of class edu.rit.util.Random, to generate the random $(x, y)$ dart coordinates. The PRNG’s seed (constructor argument) determines the sequence of random numbers the PRNG will produce. Two PRNGs initialized with the same seed will generate the same sequence; two PRNGs initialized with different seeds will generate different sequences. The user specifies the seed and the number of darts $N$ on the command line.

I used the Parallel Java 2 Library class edu.rit.util.Random, rather than the standard Java class java.util.Random, for two reasons: my PRNG class is faster than Java’s PRNG class; and my PRNG class has features useful for parallel programming that Java’s PRNG class lacks (we will look at these features later).

Listing 4.2 is the parallel $\pi$ program. It illustrates how to set up the reduction pattern. Instead of a single counter variable, I need a per-thread counter in each parallel team thread ($C_0$ through $C_3$ in Figure 4.2) plus a global counter variable to hold the result of the reduction ($C$ in Figure 4.2). In order to do the reduction, the global counter variable can no longer be type long. Instead, it is an instance of class edu.rit.pj2.vbl.LongVbl (line 14); this class performs the reduction automatically. I initialized the global counter
Chapter 4. Parallel Reduction

Figure 4.1. A dartboard for estimating $\pi$

Figure 4.2. Estimating $\pi$ sequentially and in parallel

Listing 4.1. PiSeq.java (part 1)
variable to an instance of subclass LongVbl.Sum (line 28); this subclass does a sum-reduce. (Other subclasses do other reduction operations, and you can define your own subclasses with your own reduction operations.) The per-thread counter variable is declared on line 32 and is initialized—as all per-thread variables must be—in the parallel loop’s start() method on line 36. I obtained the per-thread object by calling the parallel loop’s threadLocal() method, specifying the global variable. The threadLocal() method calls “link” the per-thread counter variables to the global counter variable and set everything up to do the reduction automatically under the hood, as I will describe shortly.

I also did away with the global PRNG variable and made it a per-thread variable, declared on line 31 and initialized on line 35. Why? Because if I left it as a global variable, all the parallel team threads would use the same PRNG object, and the threads would interfere with each other as they all accessed this object and altered its state. Making the PRNG a per-thread variable lets each thread work with its own separate PRNG object without interference from the other threads and without needing to synchronize with the other threads.

However, I had to change the way the per-thread PRNGs are seeded. Instead of initializing each one with just the seed, I initialized each one with the seed plus the thread’s rank (as returned by the rank() method). Why? Because if I seeded all the PRNGs with the same value, all the PRNGs would produce the same sequence of random numbers, and all the threads would generate the same \((x, y)\) dart coordinates. This is not what I want. I want each thread to generate different random dart coordinates. I get that by seeding the PRNGs with different values, making them produce different sequences of random numbers.

The parallel for loop starting at line 29 divides the \(N\) dart throws among the parallel team threads. I went with the default fixed schedule. Each thread increments the \texttt{item} field of its own \texttt{thrCount} per-thread variable. When the parallel loop finishes, before returning back to the main program, the contents of all the per-thread \texttt{thrCount} variables’ \texttt{item} fields are automatically added together under the hood (because, remember, these variables are of type LongVbl.Sum, which does a sum-reduce), and the result is automatically stored in the global \texttt{count} variable’s \texttt{item} field. The main program then uses \texttt{count.item} to estimate \(\pi\).

Here are runs of the sequential program and the parallel program on a four-core tardis node with \(N = \text{four billion darts}:\$
$ java pj2 debug=makespan edu.rit.pj2.example.PiSeq 142857 4000000000
pi = 4*3141556293/4000000000 = 3.141556293
Job 5 makespan 76051 msec

BIG CPU, BIG DATA
// Number of points within the unit circle.
long count;

// Main program.
public void main
(String[] args)
  throws Exception
{
  // Validate command line arguments.
  if (args.length != 2) usage();
  seed = Long.parseLong (args[0]);
  N = Long.parseLong (args[1]);

  // Set up PRNG.
  prng = new Random (seed);

  // Generate n random points in the unit square, count how many
  // are in the unit circle.
  count = 0;
  for (long i = 0; i < N; ++ i)
  {
    double x = prng.nextDouble();
    double y = prng.nextDouble();
    if (x*x + y*y <= 1.0) ++ count;
  }

  // Print results.
  System.out.printf ("pi = 4*%d/%d = %.9f%n",
                     count, N, 4.0*count/N);
}

// Print a usage message and exit.
private static void usage()
{
  System.err.println ("Usage: java pj2 "+
                      "edu.rit.pj2.example.PiSeq <seed> <N>");
  System.err.println ("<seed> = Random seed");
  System.err.println ("<N> = Number of random points");
  throw new IllegalArgumentException();
}

// Specify that this task requires one core.
protected static int coresRequired()
{
  return 1;
}

Listing 4.1. PiSeq.java (part 2)
$ java debug=makespan pj2 edu.rit.pj2.example.PiSmp 142857 \\
 4000000000

pi = 4*3141567239/4000000000 = 3.141567239
Job 6 makespan 20345 msec

The speedup was 76051 ÷ 20345 = 3.738.

Notice that, although each program’s estimate for π was within 0.001 percent of the true value, the two programs computed different estimates. This is because the two programs generated different random dart coordinates. PiSeq’s darts came from one random sequence of length four billion from a seed of 142857. PiSmp’s darts came from four random sequences, each of length one billion, each from a different seed of 142857 through 142860. For the same reason, PiSmp will compute different answers when run with different numbers of cores (threads).

For the π estimating problem, it doesn’t much matter if the sequential and parallel programs compute different answers. But for another problem involving random numbers, there might be a requirement for the parallel program to compute exactly the same answer, no matter how many cores it runs on. Such a program would have to generate the same random numbers regardless of the number of cores. This can be tricky when the random numbers are not all being generated by the same thread. We’ll look at this in the next chapter.

Under the Hood

Figure 4.3 shows how the automatic sum-reduce of the per-thread thrCount variables into the shared global count variable happens under the hood when running with eight threads. The boxes at the top show the contents of the thrCount variables’ item fields at the end of each parallel team thread’s iterations.

The reduction proceeds in a series of rounds. Here’s what each thread does in the first round:

• Thread 0 waits for thread 4 to finish its iterations, then thread 0 adds thread 4’s item field to its own item field.
• Thread 1 waits for thread 5 to finish its iterations, then thread 1 adds thread 5’s item field to its own item field.
• Thread 2 waits for thread 6 to finish its iterations, then thread 2 adds thread 6’s item field to its own item field.
• Thread 3 waits for thread 7 to finish its iterations, then thread 3 adds thread 7’s item field to its own item field.
• Threads 4 through 7 do nothing.

The above operations proceed concurrently. Notice that each thread only synchronizes with one other thread; this results in less overhead than if all the
public class PiSmp
extends Task
{
  // Command line arguments.
  long seed;
  long N;
  // Number of points within the unit circle.
  LongVbl count;
  // Main program.
  public void main
      (String[] args)
      throws Exception
  {
    // Validate command line arguments.
    if (args.length != 2) usage();
    seed = Long.parseLong (args[0]);
    N = Long.parseLong (args[1]);
    // Generate n random points in the unit square, count how many
    // are in the unit circle.
    count = new LongVbl.Sum (0);
    parallelFor (0, N - 1).exec (new LongLoop()
    {
      Random prng;
      LongVbl thrCount;
      public void start()
      {
        prng = new Random (seed + rank());
        thrCount = threadLocal (count);
      }
      public void run (long i)
      {
        double x = prng.nextDouble();
        double y = prng.nextDouble();
        if (x*x + y*y <= 1.0) ++ thrCount.item;
      }
    });
    // Print results.
    System.out.printf ("pi = 4*%d/%d = %.9f%n",
        count.item, N, 4.0*count.item/N);
  }
  // Print a usage message and exit.
  private static void usage()
  {
    System.err.println ("Usage: java pj2 "+
        "edu.rit.pj2.example.PiSmp <seed> <N>");
    System.err.println ("<seed> = Random seed");
    System.err.println ("<N> = Number of random points");
    throw new IllegalArgumentException();
  }
}

Listing 4.2. PiSmp.java (part 2)
threads had to synchronize with each other.

The second round is similar to the first round, except only threads 0 and 1 synchronize with threads 2 and 3. The third round is similar, except only thread 0 synchronizes with thread 1. After all the rounds, thread 0’s item field holds the sum of all the threads’ original item fields. The sum is then stored in the global variable’s item field. Once the reduction is complete, all the threads proceed to the end-of-loop barrier as usual.

This pattern is called a parallel reduction tree. The tree can be extended upwards to accommodate any number of threads. At each round of the reduction, half of the threads’ item fields are combined in parallel with the other half of the threads’ item fields. Thus, with \( K \) threads, the number of reduction rounds is \( \log_2 K \). The reduction tree is efficient even with large numbers of threads.

The Parallel Java 2 Library has classes in package edu.rit.pj2.vbl for doing parallel reduction on primitive types—boolean, byte, char, short, int, long, float, and double. Each of these classes implements interface Vbl in package edu.rit.pj2, which specifies the interface for a reduction variable class. You can create reduction variables for arbitrary data types, such as objects and arrays, by defining your own classes that implement interface Vbl; we’ll see an example in the next chapter.

In the PiSmp program, I declared the reduction variables to be type LongVbl, so their item fields were type long. I initialized the reduction variables with instances of class LongVbl.Sum, a subclass of LongVbl whose reduction operation is summation. I initialized the global count variable with a new instance of class LongVbl.Sum, and I initialized the per-thread thrCount variables by calling the parallel loop’s threadLocal() method. Initializing the global and per-thread variables this way links the pre-thread variables to the global variable and causes the parallel reduction to happen automatically as the parallel team threads finish the parallel for loop.

Why use the parallel reduction pattern? Chiefly to minimize the amount of thread synchronization the parallel program has to do. Each thread can manipulate its own per-thread variables without needing to synchronize at all with the other threads. At the end of the parallel loop, only a few pairwise thread synchronizations have to happen in the parallel reduction tree. Minimizing the amount of thread synchronization, and combining the per-thread results together in parallel, reduces the parallel program’s running time and improves its performance.
Points to Remember

- Use the parallel reduction pattern to combine per-thread results together into one global result.
- Make the reduction variables instances of classes that implement interface Vbl, ones having the desired data types and reduction operations.
- Initialize a parallel loop’s per-thread reduction variables by calling the threadLocal() method.
- When generating random numbers in a parallel loop, declare the PRNG as a per-thread variable.
- Initialize each thread’s per-thread PRNG with a different seed.
- Alternatively, initialize all the threads’ per-thread PRNGs with the same seed, and use the skip() method to skip past the other threads’ random numbers.

Figure 4.3. Sum-reduce parallel reduction tree