Chapter 6
Load Balancing

► 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
The Euler totient function of a number \( n \), denoted \( \Phi(n) \), is the number of numbers in the range 1 through \( n - 1 \) that are relatively prime to \( n \). Two numbers are said to be relatively prime if they have no factors in common other than 1. The Euler totient function is the foundation upon which the security of the RSA public key cryptosystem rests. An RSA public key consists of a modulus \( n \) and an exponent \( e \). The modulus is the product of two large prime numbers. The prime numbers are typically 300 or more digits long, so \( n \) is typically 600 or more digits long. If you could compute the Euler totient of an RSA public key modulus, you would be able to decrypt messages encrypted with that key. This would let you hack into the connection between a web browser and a secure web site, steal passwords and credit card numbers, and generally bring Internet commerce to a standstill. So far, however, no one has managed to invent an algorithm that computes the Euler totient of the product of two 300-digit prime numbers in a practical amount of time.

Listing 6.1 is a sequential program to compute \( \Phi(n) \). The program simply loops through every number \( i \) from 1 through \( n - 1 \), determines whether \( i \) and \( n \) are relatively prime, and increments a counter if so. The program decides whether \( i \) and \( n \) are relatively prime by computing a list of the prime factors of each number and comparing the lists to see if they have any entries in common. The program factorizes a number using trial division, similar to the algorithm in the primality testing program in Chapter 2.

For example, suppose I am computing \( \Phi(n) \) for \( n = 100 \). The prime factors of 100 are \{2, 2, 5, 5\}. Next I compute the prime factors of every number \( i \) from 1 to 99. For example, the prime factors of 35 are \{5, 7\}. Comparing the lists of factors, I see that 35 and 100 have a factor in common, namely 5. Therefore 35 and 100 are not relatively prime. On the other hand, the prime factors of 39 are \{3, 13\}; 39 and 100 have no factors in common (other than 1); so 39 and 100 are relatively prime. It turns out that the following numbers are relatively prime to 100: 1, 3, 7, 9, 11, 13, 17, 19, 21, 23, 27, 29, 31, 33, 37, 39, 41, 43, 47, 49, 51, 53, 57, 59, 61, 63, 67, 69, 71, 73, 77, 79, 81, 83, 87, 89, 91, 93, 97, and 99. There are 40 of them, so \( \Phi(100) = 40 \).

This is a horribly inefficient way to compute the Euler totient. I’m not trying to develop the world’s finest Euler totient program; I’m using this example to make a point about parallel programming. Bear with me.

To parallelize this program, note that there are no sequential dependencies in the loop on lines 24–26. The computation for each number \( i \) can be done independently of every other number. So I can change the loop to a parallel for loop and use the parallel reduction pattern to add all the per-thread counters together. Listing 6.2 is the result.

Here is what each program printed when told to compute the Euler totient of \( n = 10,000,019 \) on a four-core tardis node. Because this \( n \) is prime, every number from 1 through \( n - 1 \) is relatively prime to \( n \), so \( \Phi(n) = n - 1 \) in this case.
package edu.rit.pj2.example;
import edu.rit.pj2.Task;
import edu.rit.util.LongList;
public class TotientSeq
    extends Task
{
    long n;
    long phi;
    LongList nFactors = new LongList();
    LongList iFactors = new LongList();

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

        // Compute totient.
        phi = 0;
        factorize(n, nFactors);
        for (long i = 2; i < n; ++ i)
            if (relativelyPrime(factorize(i, iFactors), nFactors))
                ++ phi;

        // Print totient.
        System.out.printf("%d%n", phi + 1);
    }

    // Store a list of the prime factors of <I>x</I> in ascending
    // order in the given list.
    private static LongList factorize
        (long x,
         LongList list)
    {
        list.clear();
        long p = 2;
        long psqr = p*p;
        while (psqr <= x)
        {
            if (x % p == 0)
                { list.addLast (p);
                    x /= p;
                }
            else
                { p = p == 2 ? 3 : p + 2;
                    psqr = p*p;
                }
        }
        if (x != 1)
            list.addLast (x);
        return list;
    }

Listing 6.1. TotientSeq.java (part 1)
BIG CPU, BIG DATA

$ java pj2 debug=makespan edu.rit.pj2.example.TotientSeq \ 10000019
10000018
Job 9 makespan 76015 msec
$ java pj2 debug=makespan edu.rit.pj2.example.TotientSmp \ 10000019
10000018
Job 10 makespan 25293 msec

The speedup was 76015 ÷ 25293 = 3.005. Unlike the previous parallel program’s we’ve studied, the TotientSmp program’s speedup is not very close to the ideal speedup of 4. Ideally, the TotientSmp program’s running time on the four-core node should have been 76015 ÷ 4 = 19004 msec. What’s going on?

To find out, we have to look at how much time each parallel team thread spends executing its portion of the parallel loop. I modified the TotientSmp program to measure and print each thread’s running time. Look at the result:

<table>
<thead>
<tr>
<th>Thread rank</th>
<th>Running time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10780 msec</td>
</tr>
<tr>
<td>1</td>
<td>17893 msec</td>
</tr>
<tr>
<td>2</td>
<td>22249 msec</td>
</tr>
<tr>
<td>3</td>
<td>25602 msec</td>
</tr>
</tbody>
</table>

Ideally, each thread should spend the same amount of time in the parallel loop. But we see they do not; some take more time than others, some take less. Because the parallel loop does not finish until the longest-running thread finishes, the program’s running time ends up being larger and its speedup smaller than they should be.

This situation, where the parallel team threads take different amounts of time to execute, is called an unbalanced load. An unbalanced load is undesirable; it causes the parallel program to take more time than necessary. Rather, we want the program to have a balanced load, where every thread takes about the same amount of time.

But why is the TotientSmp program’s load unbalanced? After all, the parallel for loop is using the default schedule, namely a fixed schedule, which gives each parallel team thread an equal portion of the loop iterations. However, in the TotientSmp program, the code executed in each loop iteration on lines 36–37 takes a different amount of time in each loop iteration. Why? Because as the number i increases, the factorize() method takes longer and longer to compute the prime factors of i. Thus, the higher-ranked threads—the ones that factorize the larger values of i—take longer than the lower-ranked threads. This leads to the unbalanced load. In the previous chapters’ parallel programs, each parallel for loop iteration took the same amount of time; so the load was inherently balanced, and using the default fixed schedule resulted in close-to-ideal speedups.

How can we get a balanced load in the TotientSmp program? We have to use a different parallel for loop schedule. Instead of dividing the parallel for loop iterations into four large chunks (one chunk for each parallel team
private static boolean relativelyPrime
    (LongList xFactors,
     LongList yFactors)
    {
        int xSize = xFactors.size();
        int ySize = yFactors.size();
        int ix = 0;
        int iy = 0;
        long x, y;
        while (ix < xSize && iy < ySize)
        {
            x = xFactors.get (ix);
            y = yFactors.get (iy);
            if (x == y) return false;
            else if (x < y) ++ ix;
            else ++ iy;
        }
        return true;
    }

    // Print a usage message and exit.
    private static void usage()
    {
        System.err.println ("Usage: java pj2 " +
            "edu.rit.pj2.example.TotientSeq <n>");
        throw new IllegalArgumentException();
    }

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

Listing 6.1. TotientSeq.java (part 2)
thread) as the fixed schedule does, let’s divide the loop iterations into many small chunks with, say, 1000 iterations in each chunk. Then let the threads execute chunks in a dynamic fashion. Each thread starts by executing one chunk. When a thread finishes its chunk, it executes the next available chunk. When all chunks have been executed, the parallel for loop finishes. This way, some threads execute fewer longer-running chunks, other threads execute more shorter-running chunks, the threads finish at roughly the same time, and the load is balanced. This is called a dynamic schedule.

You can specify the parallel for loop schedule and chunk size on the pj2 command line by including the schedule and chunk parameters. These override the default fixed schedule. Here is the same TotientSmp program run on tardis, this time with a dynamic schedule and a chunk size of 1000 iterations:

```
$ java pj2 debug=makespan schedule=dynamic chunk=1000 \
   edu.rit.pj2.example.TotientSmp 10000019 10000018
Job 11 makespan 19178 msec
```

This time the speedup was \( \frac{76015}{19178} = 3.964 \), very close to an ideal speedup. The dynamic schedule has indeed balanced the load, as is evident from the parallel team threads’ individual running times:

```
<table>
<thead>
<tr>
<th>Thread rank</th>
<th>Running time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>19154 msec</td>
</tr>
<tr>
<td>1</td>
<td>19155 msec</td>
</tr>
<tr>
<td>2</td>
<td>19155 msec</td>
</tr>
<tr>
<td>3</td>
<td>19156 msec</td>
</tr>
</tbody>
</table>
```

With a dynamic schedule, there’s a tradeoff. If the chunk size is too large, the load can become unbalanced again, like a fixed schedule. The unbalanced load can result in a longer running time. However, if the chunk size is too small, there will be extra overhead in the program, as the parallel loop has to generate and feed more chunks to the parallel team threads. This extra overhead can also result in a longer running time. It’s not always apparent what the best chunk size should be for a dynamic schedule.

As an alternative, you can specify a proportional schedule. Instead of specifying chunks of a certain size, you specify a certain number of chunks. The number of chunks is the number of threads times a chunk factor. The set of loop iterations is partitioned into that many equal-sized chunks, and the threads execute these chunks in a dynamic fashion. As the number of threads increases, the number of chunks also increases proportionally, and the chunks become smaller. Here is the same TotientSmp run on tardis with a proportional schedule:

```
$ java pj2 debug=makespan schedule=proportional \
   edu.rit.pj2.example.TotientSmp 10000019 10000018
Job 12 makespan 19775 msec
```
Listing 6.2. TotientSmp.java (part 2)
For the above run on the four-core tardis node, the loop index range was partitioned into 40 chunks—4 threads times the default chunk factor of 10. The speedup was 3.844; better than the fixed schedule, not quite as good as the dynamic schedule.

As another alternative, you can specify a guided schedule. Like a dynamic schedule, a guided schedule divides the parallel for loop iterations into many smaller chunks. However, the chunks are not all the same size. Earlier chunks have more iterations; later chunks have fewer iterations. This tends to balance the load automatically without needing to specify the chunk size. Here is the same TotientSmp program run on tardis with a guided schedule:

```
$ java pj2 debug=makespan schedule=guided \
edu.rit.pj2.example.TotientSmp 10000019
10000018
Job 13 makespan 18931 msec
```

The speedup was 4.015, even better than the dynamic schedule—in fact, essentially an ideal speedup. (That it is slightly greater than 4 might be due to random measurement error.) For a guided schedule, the chunk parameter gives the minimum chunk size; if omitted, the default is 1.

When a parallel program needs load balancing, you should experiment with different parallel for loop schedules, chunk sizes, and chunk factors on typical inputs to determine the schedule that yields the smallest running time. If you choose, you can then hard-code the schedule into the program; this overrides the default schedule and any schedule specified on the pj2 command line. To get a dynamic schedule with a chunk size of 1000, write:

```
parallelFor(lb,ub) .schedule(dynamic) .chunk(1000) ...
```

To get a proportional schedule with a chunk factor of 100, write:

```
parallelFor(lb,ub) .schedule(proportional) .chunk(100) ...
```

To get a guided schedule with the default minimum chunk size, write:

```
parallelFor(lb,ub) .schedule(guided) ...
```

### Under the Hood

Figure 6.1 shows how various schedules partition the iterations of a parallel for loop into chunks. The loop has $N = 100$ iterations and is being executed by a parallel team with $K = 4$ threads.

- A fixed schedule partitions the iterations into $K$ chunks, each of size $N/K$, and assigns one chunk to each team thread. If $N$ is not evenly divisible by $K$, the final chunk has fewer iterations than the other chunks. A leapfrog schedule is similar, except each team thread increments the loop index by $K$ on each iteration instead of by 1.
Listing 6.2. TotientSmp.java (part 3)

```java
(ListLong xFactors,
 LongList yFactors)
{
    int xSize = xFactors.size();
    int ySize = yFactors.size();
    int ix = 0;
    int iy = 0;
    long x, y;
    while (ix < xSize && iy < ySize)
    {
        x = xFactors.get (ix);
        y = yFactors.get (iy);
        if (x == y) return false;
        else if (x < y) ++ ix;
        else ++ iy;
    }
    return true;
}

// Print a usage message and exit.
private static void usage()
{
    System.err.println ("Usage: java pj2 " +
    "edu.rit.pj2.example.TotientSmp <n>");
    throw new IllegalArgumentException();
}
```

Figure 6.1. Chunk sizes for 100 iterations and four threads
• A dynamic schedule partitions the iterations into chunks of a fixed size (5 iterations in this example). The default chunk size is 1. If \( N \) is not evenly divisible by the chunk size, the final chunk has fewer iterations than the other chunks. Each chunk is assigned to a team thread at the beginning of the loop and whenever a team thread finishes its previous chunk.

• A proportional schedule is similar to a dynamic schedule, except it partitions the iterations into a fixed number of chunks rather than a fixed size of chunks. The number of chunks is equal to a chunk factor times \( K \); the default chunk factor is 10. Thus, as \( K \) increases, the number of chunks also increases proportionally. The chunk size is \( \frac{N}{C \cdot K} \), where \( C \) is the chunk factor. If \( N \) is not evenly divisible by \( C \cdot K \), the final chunk has fewer iterations than the other chunks.

• A guided schedule is similar to a dynamic schedule, except it determines the size of each chunk on the fly. Each chunk’s size is half the number of remaining iterations divided by \( K \). If this is less than the specified minimum chunk size (default one iteration), the chunk size is the minimum chunk size. Earlier chunks have more iterations, later chunks have fewer iterations. With \( N = 100 \) and \( K = 4 \), the guided schedule’s chunk sizes are 12, 11, 9, 8, 7, 6, and so on.

**Points to Remember**

• If different iterations of a parallel for loop can take different amounts of time to execute, use a dynamic, proportional, or guided schedule to balance the load.

• Run the parallel program on typical inputs with various schedules, chunk sizes, and chunk factors to determine the schedule that yields the smallest overall running time.