Chapter 7
Parallel Loop Schedules

► Part I. Preliminaries
▼ Part II. Tightly Coupled Multicore
  Chapter 6. Parallel Loops
  Chapter 7. Parallel Loop Schedules
  Chapter 8. Parallel Reduction
  Chapter 9. Reduction Variables
  Chapter 10. Load Balancing
  Chapter 11. Overlapping
  Chapter 12. Sequential Dependencies
  Chapter 13. Strong Scaling
  Chapter 14. Weak Scaling
  Chapter 15. Exhaustive Search
  Chapter 16. Heuristic Search
  Chapter 17. Parallel Work Queues
► Part III. Loosely Coupled Cluster
► Part IV. GPU Acceleration
► Part V. Map-Reduce
► Appendices
Bitcoin (bitcoin.org) is an open-source, peer-to-peer, digital currency system. Bitcoins are created by “mining,” reminiscent of how the 1849 Gold Rush prospectors panned for gold. Our next program example implements a simplified version of bitcoin mining. You specify a coin ID, which is just a string of hexadecimal digits, along with a number \( n \). The program then finds a 64-bit nonce such that the digest has \( n \) leading zero bits. The digest is computed by concatenating the coin ID and the nonce, applying the SHA-256 cryptographic hash function to that, and applying SHA-256 to the result:

\[
digest = \text{SHA-256 (SHA-256 (coin ID || nonce))}
\]

The digest is a 256-bit number. An example will clarify what the program does:

```
$ java pj2 edu.rit.pj2.example.MineCoinSeq fedcba9876543210 16
Coin ID = fedcba9876543210
Nonce   = 0000000000019c48
Digest  = 0000dec325d10670f7abab011a37a0f2dcdaba02a6b65a6a59e7eec7ba77448c
```

For the coin ID fedcba9876543210, the program found that the nonce 19c48 yielded a digest with 16 leading zero bits (4 leading zero hexadecimal digits). How did it find this nonce? By brute force search. The program simply started with nonce 0, computed the digest, incremented the nonce, and repeated until the digest had \( n \) leading zero bits. Like a gold miner, the program had to pan through 105,544 worthless nonces until it hit that golden nonce.

How many nonces must the program examine in general before finding a golden nonce? The output of a cryptographic hash function looks like a random number; each bit is a zero or a one with probability \( 1/2 \), independent of all the other bits. Therefore, the probability that the \( n \) leading bits are all zeroes is \( 1/2^n \), and we expect to examine \( 2^n \) nonces on the average before we find a golden nonce. Even for small \( n \) values like 24 or 32, that’s a lot of nonces. A parallel program should let us find a golden nonce more quickly.
package edu.rit.pj2.example;
import edu.rit.crypto.SHA256;
import edu.rit.pj2.Task;
import edu.rit.util.Hex;
import edu.rit.util.Packing;
public class MineCoinSeq
extends Task
{
    // Command line arguments.
    byte[] coinId;
    int N;

    // Mask for leading zeroes.
    long mask;

    // For computing hash digests.
    byte[] coinIdPlusNonce;
    SHA256 sha256;
    byte[] digest;

    // Main program.
    public void main
    (String[] args)
    throws Exception
    {
        // Validate command line arguments.
        if (args.length != 2) usage();
        coinId = Hex.toByteArray (args[0]);
        N = Integer.parseInt (args[1]);
        if (1 > N || N > 63) usage();

        // Set up mask for leading zeroes.
        mask = ~((1L << (64 - N)) - 1L);

        // Set up for computing hash digests.
        coinIdPlusNonce = new byte [coinId.length + 8];
        System.arraycopy (coinId, 0, coinIdPlusNonce, 0,
                         coinId.length);
        sha256 = new SHA256();
        digest = new byte [sha256.digestSize()];

        // Try all nonces until the digest has N leading zero bits.
        for (long nonce = 0L; nonce <= 0x7FFFFFFFFFFFFFFFL; ++ nonce)
        {
            // Test nonce.
            Packing.unpackLongBigEndian (nonce, coinIdPlusNonce,
                                         coinId.length);
            sha256.hash (coinIdPlusNonce);
            sha256.digest (digest);
            sha256.hash (digest);
            sha256.digest (digest);
            if ((Packing.packLongBigEndian (digest, 0) & mask) == 0L)
            {
                // Print results.
                System.out.printf ("Coin ID = %s%n",
                                   Hex.toString (coinId));
                System.out.printf ("Nonce   = %s%n",
                                   Hex.toString (nonce));
            }
        }
    }
}

Listing 7.1. MineCoinSeq.java (part 1)
If we want the program to finish in a reasonable amount of time, the desired number of leading zero bits in the digest (24, say) is going to have to be much smaller than the number of bits in the nonce (64). Therefore, by the pigeonhole principle—that if you put $x$ pigeons into $y$ pigeonholes, $x > y$, some pigeonholes will have more than one pigeon—there will be many nonce values all of which yield a digest with the desired number of leading zero bits. We’d expect there to be about $2^{64} \div 2^{24}$, or $2^{40}$, golden nonces that yield 24 leading zero bits in the digest. The program will stop as soon as it finds the first of these golden nonces.

Program MineCoinSeq (Listing 7.1) is a sequential program that solves the preceding problem. It begins by declaring several global variables on lines 9–19. These are actually fields of the Task subclass, so they are in scope for all of the task’s methods as well as any inner classes. Lines 27–30 extract the coin ID and $n$ from the command line. Line 33 computes a mask with the $n$ most significant bits set to 1 and the other bits set to 0; the bits that are 1 in the mask correspond to the bit positions that must be 0 in the digest. Lines 36–38 set up a byte array that will be input to the hash function, with the coin ID at the beginning plus eight bytes at the end to hold the nonce. Line 39 creates an object to compute the SHA-256 hash function, using class edu.rit.crypto.SHA256. Line 40 sets up a byte array to hold the digest.

Lines 43–63 are the heart of the computation. The loop iterates over all nonces from 0 to $2^{63} – 1$. The loop body unpacks the 64-bit nonce into the final eight bytes of the byte array, computes the digest of the coin ID and nonce, extracts the first eight bytes of the digest as a long, and bitwise-ands that with the mask. If the result is zero, then the $n$ most significant bits of the digest are all 0s. (The $n$ most significant bits of the digest were anded with 1s in the mask, so they stayed intact. It doesn’t matter what the other bits of the digest are; they were anded with 0s in the mask, so they became 0s.) When the program finds this golden nonce, it prints the results and breaks out of the loop without testing any further nonces—an early loop exit.

Here’s a run of the sequential program on a tardis node, with a random coin ID and a larger $n$:

```
$ java pj2 debug=makespan edu.rit.pj2.example.MineCoinSeq \
  b3e5da601135706f 24
Coin ID = b3e5da601135706f
Nonce   = 0000000000f6fa72
Digest  = 000000f4a186e571af5e0910fc48a0ed87078f7337016d4d406cf3a05b328e8
Job 1 makespan 31746 msec
```

Now let’s go parallel with program MineCoinSmp (Listing 7.2). As I did with the primality testing program in Chapter 6, I’ll change the for loop to a parallel for loop. But notice that the loop body code refers to the global variables coinIdPlusNonce, sha256, and digest. In program MineCoinSmp,
System.out.printf("Digest = %s%n", Hex.toString(digest));
break;
}
}

// Print a usage message and exit.
private static void usage()
{
    System.err.println("Usage: java pj2 " +
        "edu.rit.pj2.example.MineCoinSeq <coinid> <N>");
    System.err.println("<coinid> = Coin ID (hexadecimal)");
    System.err.println("<N> = Number of leading zero bits " +
        "(1 .. 63)");
    terminate(1);
}

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

Listing 7.1. MineCoinSeq.java (part 2)

package edu.rit.pj2.example;
import edu.rit.crypto.SHA256;
import edu.rit.pj2.LongLoop;
import edu.rit.pj2.Schedule;
import edu.rit.pj2.Task;
import edu.rit.util.Hex;
import edu.rit.util.Packing;
public class MineCoinSmp
    extends Task
{
    // Command line arguments.
    byte[] coinId;
    int N;

    // Mask for leading zeroes.
    long mask;

    // Main program.
    public void main
        (String[] args)
        throws Exception
    {
        // Validate command line arguments.
        if (args.length != 2) usage();
        coinId = Hex.toByteArray(args[0]);
        N = Integer.parseInt(args[1]);
        if (1 > N || N > 63) usage();

        // Set up mask for leading zeroes.
        mask = ~((1L << (64 - N)) - 1L);

    Listing 7.2. MineCoinSmp.java (part 1)
there will be multiple threads executing copies of the loop body concurrently. If the threads all manipulate the same global variables, the threads will interfere with each other; when one thread changes the state of a shared global variable, that will wipe out another thread’s changes, and the program will not find the correct answer. To avoid this, I want each thread to have its own separate copy of each of these variables. These are called thread-local variables. However, the other variables—coinId, N, and mask—can remain shared global variables, because the threads will only read them, not change them. These are called shared global WORM variables—variables that are shared by all the threads, Written Once, and Read Many times.

Program MineCoinSmp starts out the same as program MineCoinSeq, with declarations for the shared global variables coinId, N, and mask. The main program code is the same, up to the loop at line 32, which becomes a parallel for loop. There’s a call to the parallel for loop’s schedule() method, which I’ll gloss over for now. The loop body is in the inner LongLoop subclass. (The loop body uses class LongLoop, which has a loop index of type long, rather than class Loop, which has a loop index of type int.) This time, I declare the coinIdPlusNonce, sha256, and digest variables as fields of the inner LongLoop subclass, which makes them thread-local variables. Why? Recall that each thread in the parallel for loop’s team works with its own separate copy of the loop body object; thus, each team thread ends up with its own separate copies of the fields (thread-local variables).

The thread-local variables are not initialized as part of their declarations. Rather, the thread-local variables must be initialized in the loop body’s start() method (lines 40–48). Why? Because when each team thread clones the loop body object, the clone() method creates a copy of the loop body object itself, but then assigns the fields of the original to the fields of the copy. At this point, every team thread’s loop body’s fields refer to the same objects as the original loop body’s fields. To ensure that each team thread gets its own separate copies of the thread-local variables, the fields must be re-initialized after cloning the loop body object. That is the purpose of the start() method. After cloning the loop body object, each team thread calls the start() method, once only, to initialize its own loop body object’s fields. Each team thread then proceeds to call the run() method repeatedly to perform the loop iterations.

The code for one loop iteration is in the loop body object’s run() method, with the nonce loop index passed in as an argument. But this time the variables coinIdPlusNonce, sha256, and digest are thread-local variables. Each team thread can read and update these variables without needing to synchronize with the other team threads, because each team thread has its own copies.

After one team thread finds a golden nonce and prints its results, I want to do an early loop exit and terminate the program. But I can’t do a break
// Try all nonces until the digest has N leading zero bits.
parallelFor (0L, 0x7FFFFFFFFFFFFFFFL)
.schedule (leapfrog) .exec (new LongLoop()
{
    // For computing hash digests.
    byte[] coinIdPlusNonce;
    SHA256 sha256;
    byte[] digest;

    public void start() throws Exception
    {
        // Set up for computing hash digests.
        coinIdPlusNonce = new byte [coinId.length + 8];
        System.arraycopy (coinId, 0, coinIdPlusNonce, 0,
            coinId.length);
        sha256 = new SHA256();
        digest = new byte [sha256.digestSize()];
    }

    public void run (long nonce)
    {
        // Test nonce.
        Packing.unpackLongBigEndian (nonce, coinIdPlusNonce,
            coinId.length);
        sha256.hash (coinIdPlusNonce);
        sha256.digest (digest);
        sha256.hash (digest);
        sha256.digest (digest);
        if ((Packing.packLongBigEndian (digest, 0) & mask)
            == 0L)
        {
            // Print results.
            System.out.printf ("Coin ID = %s\n",
                Hex.toString (coinId));
            System.out.printf ("Nonce   = %s\n",
                Hex.toString (nonce));
            System.out.printf ("Digest  = %s\n",
                Hex.toString (digest));
            stop();
        }
    }

    // Print a usage message and exit.
    private static void usage()
    {
        System.err.println ("Usage: java pj2 "+
            "edu.rit.pj2.example.MineCoinSmp <coinid> <N>\");
        System.err.println ("<coinid> = Coin ID (hexadecimal)\");
        System.err.println ("<N> = Number of leading zero bits " +
            "(1 .. 63)\");
        terminate (1);
    }
}

Listing 7.2. MineCoinSmp.java (part 2)
statement as in the sequential program, because there is no for statement in the parallel program. Instead, I call the stop() method (line 69). This tells the parallel for loop to stop iterating in all the team threads. After finishing its current iteration, each team thread immediately proceeds to the end-of-loop barrier, and then the program goes on to execute the code after the parallel for loop.

Let’s go back to the schedule() method call on line 33 and explain what it does. Every parallel for loop has a schedule that determines the loop indexes that each team thread will perform. In other words, the schedule determines how the work of the for loop is divided among the team threads.

If you don’t specify a schedule for a parallel for loop, the default is to use a fixed schedule. A fixed schedule divides the loop index range into $K$ consecutive subranges, where $K$ is the number of team threads. Each subrange is the same size, except possibly the last. (If the loop index range is not evenly divisible by $K$, the last subrange will include only the leftover indexes.) For example, suppose the loop index goes from 1 to 100. Here’s how a fixed schedule would divide the loop index range among different numbers of team threads:

- $K = 1$
  - Team thread rank 0 does indexes 1, 2, 3, \ldots, 100 (100 indexes)
- $K = 2$
  - Team thread rank 0 does indexes 1, 2, 3, \ldots, 50 (50 indexes)
  - Team thread rank 1 does indexes 51, 52, 53, \ldots, 100 (50 indexes)
- $K = 3$
  - Team thread rank 0 does indexes 1, 2, 3, \ldots, 34 (34 indexes)
  - Team thread rank 1 does indexes 35, 36, \ldots, 68 (34 indexes)
  - Team thread rank 2 does indexes 69, 70, 71, \ldots, 100 (32 indexes)
- $K = 4$
  - Team thread rank 0 does indexes 1, 2, 3, \ldots, 25 (25 indexes)
  - Team thread rank 1 does indexes 26, 27, 28, \ldots, 50 (25 indexes)
  - Team thread rank 2 does indexes 51, 52, 53, \ldots, 75 (25 indexes)
  - Team thread rank 3 does indexes 76, 77, 78, \ldots, 100 (25 indexes)

Thus, with a fixed schedule, the work of the parallel for loop is divided as equally as possible among the team threads, and each team thread does a contiguous range of loop indexes.

Program MineCoinSmp’s parallel for loop goes through $2^{63}$ indexes, from 0 to $2^{63} – 1 = 9223372036854775807$ (disregarding the early loop exit). If I were to use the default fixed schedule, and if I were to run the program on a tardis node with 12 cores, look at the indexes each team thread would do:
Chapter 7. Parallel Loop Schedules

• Rank 0 does 0 – 768614336404564650
• Rank 1 does 768614336404564651 – 1537228672809129301
• Rank 2 does 1537228672809129302 – 2305843009213693952
• Rank 3 does 2305843009213693953 – 3074457345618258603
• Rank 4 does 3074457345618258604 – 3843071682022823254
• Rank 5 does 3843071682022823255 – 4611686018427387905
• Rank 6 does 4611686018427387906 – 5380300354831952556
• Rank 7 does 5380300354831952557 – 6148914691236517207
• Rank 8 does 6148914691236517208 – 6917529027641081858
• Rank 9 does 6917529027641081859 – 7686143364045646509
• Rank 10 does 7686143364045646510 – 8454757700450211160
• Rank 11 does 8454757700450211161 – 9223372036854775807

This is not what I want. In effect, each team thread would be doing a separate golden nonce search within a different index range. Thread rank 0 would find the smallest golden nonce starting from nonce 0, which would be the same as the one the sequential program found. But only one thread, thread rank 0, would be searching this range of indexes, so the parallel program would take the same amount of time as the sequential program to find this golden nonce. The other team threads would search other index ranges, so I would expect each team thread to do an average of $2^n$ iterations before finding some golden nonce, the same number of iterations as the sequential program. Thus, the whole parallel program would take the same amount of time as the sequential program, and I would not experience any speedup!

Instead, I want the team threads to do loop indexes this way:

• Rank 0 does 0, 12, 24, 36, . . .
• Rank 1 does 1, 13, 25, 37, . . .
• Rank 2 does 2, 14, 26, 38, . . .
• Rank 3 does 3, 15, 27, 39, . . .
• Rank 4 does 4, 16, 28, 40, . . .
• Rank 5 does 5, 17, 29, 41, . . .
• Rank 6 does 6, 18, 30, 42, . . .
• Rank 7 does 7, 19, 31, 43, . . .
• Rank 8 does 8, 20, 32, 44, . . .
• Rank 9 does 9, 21, 33, 45, . . .
• Rank 10 does 10, 22, 34, 46, . . .
• Rank 11 does 11, 23, 35, 47, . . .

Now all the team threads are searching the same index range starting at 0 and going to 9223372036854775807, but in parallel. As soon as any team thread finds the first golden nonce, the program terminates. The work of searching
this one index range is divided equally among the team threads, so ideally the parallel program should yield a speedup factor of 12.

This parallel for loop schedule is called a **leapfrog schedule**. The loop indexes are allocated in a round robin fashion among the $K$ team threads. Each team thread does a series of **noncontiguous** loop indexes, with each index equaling the previous index plus $K$.

To get a leapfrog schedule in the MineCoinSmp program, I called the parallel for loop's `schedule()` method on line 33, passing `leapfrog` as the argument, before actually executing the loop. The possible schedules, listed in enum `edu.rit.pj2.Schedule`, are `fixed` (the default), `leapfrog`, `dynamic`, `proportional`, and `guided`. Program PrimeSmp in Chapter 6 uses a fixed schedule; program MineCoinSmp uses a leapfrog schedule; we will see examples of dynamic, proportional, and guided schedules in later chapters.

Here's a run of the parallel program on on a 12-core tardis node, with the same arguments as the previous sequential program run.

```bash
$ java pj2 debug=makespan edu.rit.pj2.example.MineCoinSmp \
   b3e5da601135706f 24
Coin ID = b3e5da601135706f
Nonce   = 0000000000f6fa72
Digest  = 000000f4a186e571af5e0910fc48a0ed87078f7337016d4d406cf 
   c3a05b328e8
Job 1 makespan 2853 msec
```

The parallel program found the same golden nonce as the sequential program, but it took only 2.9 seconds instead of 31.7 seconds—a speedup factor of $31746 \div 2853 = 11.127$.

**Under the Hood**

Figure 7.1 shows in more detail what happens when the MineCoinSmp program executes the parallel for loop on a parallel computer with 12 cores. (The `main()` method is not shown.) When the main thread calls the parallel for loop's `exec()` method, the main thread blocks and the team threads unblock. Each of the team threads creates its own copy of the loop object, calls the loop object's `start()` method which initializes the thread-local variables, and calls the loop object's `run()` method multiple times with the loop indexes dictated by the parallel for loop's schedule. When one of the team threads doing one of the `run()` invocations—team thread rank 1 calling `run(25)` in this example—finds a golden nonce, that team thread prints the nonce, calls the loop object’s `stop()` method, and proceeds to the barrier. After the loop has been stopped, as each team thread finishes its current invocation of the `run()` method, each team thread proceeds to the barrier without calling the `run()` method further. When all the team threads have arrived at
Figure 7.1. Parallel for loop execution flow, leapfrog schedule
the barrier, the team threads block, the main thread unblocks, and the main thread resumes executing the main program.

**Points to Remember**

- Declare variables as shared global WORM variables (fields of the Task subclass) if the parallel for loop team threads will read them but not alter them.
- Thread synchronization is not needed when reading shared global WORM variables.
- Declare variables as thread-local variables (fields of the loop body subclass) if the parallel for loop team threads will read and alter them.
- *Do not initialize thread-local variables as part of their declarations.* Initialize thread-local variables in the `start()` method.
- Thread synchronization is not needed when reading and writing thread-local variables.
- Decide which kind of schedule a parallel for loop needs, and specify it by calling the `schedule()` method if necessary.
- Use the `stop()` method to do an early exit from a parallel for loop.