CHAPTER 7

Massively Parallel Problems, Part 2

in which we convert the sequential program for a massively parallel cryptographic problem into an SMP parallel program; and we learn how to apply Parallel Java's SMP parallel programming features

CHAPTER 7 Massively Parallel Problems, Part 2

7.1 AES Key Search Parallel Program Design

Now that we've been introduced to the constructs for writing SMP parallel programs in Parallel Java, let's return to the AES partial key search problem from Chapter 5. In that chapter, we built a sequential program that found the complete encryption key, given a plaintext, the corresponding ciphertext, and the partial key. Now we'll build an SMP parallel program that does the same thing.

The first step in designing the parallel version is to identify the computational code that will be executed in parallel. That's easy; it is the for loop in the middle of the main program. As we have already remarked, because this is a massively parallel problem, each computation (loop iteration) is independent of every other computation, and the iterations can all be done in parallel. Thus, the for loop will become a parallel for loop inside a parallel region.

The next step is to analyze the program's variables, decide which ones will and will not be shared by the parallel team threads, and decide where to declare each variable.

- `plaintext`, `ciphertext`, `partialkey`, `n`—A program's command-line arguments are typically used throughout the program. They will always be shared variables.
- `keylsbs`—This holds the least significant 32 bits of the partial key. It is written by the main program during the setup phase and read (but not written) by each thread during the parallel for loop. Because it is accessed both by the main program and by the threads, it will be a shared variable.
- `maxcounter`—This is the upper bound for the loop counter that ranges over all possible values for the missing key bits. It is written by the main program and read (but not written) during the parallel for loop. It, too, will be a shared variable.
- `foundkey`—This holds a copy of the key that was found to encrypt the plaintext correctly. It is written by one of the threads during the parallel for loop and read by the main program during the cleanup phase when the results are printed. It will be a shared variable.
- `trialkey`—This holds the complete key that the current loop iteration is using for its trial encryption. While the most significant bits of the trial key remain constant, the least significant bits are different on every loop iteration. Therefore, `trialkey` must be a per-thread variable. Each thread must have its own copy so that one thread will not overwrite another thread's trial key.

- `trialciphertext`—This receives the result of the current loop iteration's trial encryption. Again, this will be different on every loop iteration. It, too, will be a per-thread variable, so that one thread will not overwrite another thread's trial ciphertext.
- `cipher`—This is the AES block cipher object that does the actual encryption. It uses a different key on each loop iteration. It, too, will be a per-thread variable, so that one thread will not change the key of another thread's cipher object.
- `t1`, `t2`—These are used solely within the main program for timing measurements. They will remain local variables of the main program.

Now we need to take a second look at each of the shared variables and decide whether the threads can conflict with each other when accessing the shared variables. If a conflict is possible, we have to decide how to synchronize the threads. We don't have to worry about the per-thread variables, because only one thread will ever access those.

- `plaintext`, `ciphertext`, `partialkey`, `n`—These are **write once, read many (WORM)** variables. Once initialized from the command-line arguments, they are never written again, only read. Because multiple threads reading a shared variable do not conflict with each other, no synchronization is needed for these variables.
- `keylsbs`—WORM variable. No synchronization is needed.
- `maxcounter`—WORM variable. No synchronization is needed.
- `foundkey`—Because there are 2^{256} possible AES keys, but only 2^{128} possible ciphertext blocks, it must be the case that different keys yield the same ciphertext block for a given plaintext block. Therefore, it is possible for different threads to find a correct key and store it in the `foundkey` variable. Consequently, write-write conflicts are possible, and thread synchronization is needed. After the parallel region has finished, when the main program prints the contents of `foundkey`, only the main thread is executing; so no conflicts are possible and no synchronization is needed at this point.
- The remaining variables are per-thread variables or main program local variables and need no synchronization.

We have reached the conclusion that *this* program does not need to synchronize the threads when accessing the shared variables, except when writing the `foundkey` variable. It's important to emphasize that we are not merely omitting the synchronization out of laziness. We have carefully analyzed the program, decided where synchronization is and is not needed, and deliberately omitted any unnecessary synchronization, thereby eliminating the overhead that goes with it. When designing a parallel program, you must *always* analyze how your variables are accessed.

To prevent write-write conflicts over the `foundkey` variable, each thread will make a copy of the correct key in a temporary byte array, and then will store a reference to this byte array in the `foundkey`

variable. Reads and writes of an array reference variable in Java are guaranteed to be **atomic**. If multiple threads try to read or write the variable simultaneously, the JVM ensures that each read or write operation finishes before the next read or write operation begins. Thus, the JVM itself synchronizes multiple threads writing the `foundkey` variable. It's important to emphasize that this synchronization happens only when writing the array *reference,* not the array *elements.* That's why each thread makes its own copy of the array elements first, and afterward writes the array reference into the `foundkey` variable.

Reads and writes of the following Java primitive types are guaranteed to be atomic: `boolean`, `byte`, `char`, `short`, `int`, and `float`. Reads and writes of object and array references are also guaranteed to be atomic. Reads and writes of the types `long` and `double` are not guaranteed to be atomic. In addition, *updates* of any type of variable—where the old value is read and a new value is computed and written back—are not guaranteed to be atomic. SMP parallel programs where multiple threads use `long` or `double` shared variables, or where multiple threads update shared variables, must synchronize the threads when they access such variables.

7.2 AES Key Search Parallel Program Code

Taking the foregoing design considerations into account, here is the code for the SMP parallel version of the AES key search program, class FindKeySmp.

```
package edu.rit.smp.keysearch;
import edu.rit.crypto.blockcipher.AES256Cipher;
import edu.rit.pj.IntegerForLoop;
import edu.rit.pj.ParallelRegion;
import edu.rit.pj.ParallelTeam;
import edu.rit.util.Hex;
public class FindKeySmp
   {
```

As in the sequential version, we declare the shared variables as static fields of the main program class.

```
   // Command line arguments.
   static byte[] plaintext;
   static byte[] ciphertext;
   static byte[] partialkey;
   static int n;

   // The least significant 32 bits of the partial key.
   static int keylsbs;

   // The maximum value for the missing key bits counter.
   static int maxcounter;
```

```
// The complete key.
static byte[] foundkey;

/**
 * AES partial key search main program.
 */
public static void main
   (String[] args)
   throws Exception
   {
   // Start timing.
   long t1 = System.currentTimeMillis();
```

In the main program's setup phase, we initialize the shared variables.

```
   // Parse command line arguments.
   if (args.length != 4) usage();
   plaintext = Hex.toByteArray (args[0]);
   ciphertext = Hex.toByteArray (args[1]);
   partialkey = Hex.toByteArray (args[2]);
   n = Integer.parseInt (args[3]);

   // Make sure n is not too small or too large.
   if (n < 0)
      {
      System.err.println ("n = " + n + " is too small");
      System.exit (1);
      }
   if (n > 30)
      {
      System.err.println ("n = " + n + " is too large");
      System.exit (1);
      }

   // Set up shared variables for doing trial encryptions.
   keylsbs =
      ((partialkey[28] & 0xFF) << 24) |
      ((partialkey[29] & 0xFF) << 16) |
      ((partialkey[30] & 0xFF) <<  8) |
      ((partialkey[31] & 0xFF)      );
   maxcounter = (1 << n) - 1;
```

Now comes the computational heart of the program. We set up a parallel team, which executes a parallel region, which in turn executes a parallel for loop with the index going from 0 to $2^n - 1$ (`maxcounter`), inclusive.

```
// Do trial encryptions in parallel.
new ParallelTeam().execute (new ParallelRegion()
   {
   public void run() throws Exception
      {
      execute (0, maxcounter, new IntegerForLoop()
         {
```

We declare the per-thread variables as instance fields of the parallel for loop subclass.

```
            // Thread local variables.
            byte[] trialkey;
            byte[] trialciphertext;
            AES256Cipher cipher;
```

We initialize the per-thread variables in the parallel for loop's `start()` method.

```
            // Set up thread local variables.
            public void start()
               {
               trialkey = new byte [32];
               System.arraycopy
                  (partialkey, 0, trialkey, 0, 32);
               trialciphertext = new byte [16];
               cipher = new AES256Cipher (trialkey);
               }
```

The sequential program's loop body ends up in the parallel for loop's `run()` method.

```
            // Try every possible combination of low-order key
            // bits.
            public void run (int first, int last)
               {
               for (int counter = first; counter <= last;
                        ++ counter)
                  {
                  // Fill in low-order key bits.
                  int lsbs = keylsbs | counter;
                  trialkey[28] = (byte) (lsbs >>> 24);
```

```
                        trialkey[29] = (byte) (lsbs >>> 16);
                        trialkey[30] = (byte) (lsbs >>>  8);
                        trialkey[31] = (byte) (lsbs       );

                        // Try the key.
                        cipher.setKey (trialkey);
                        cipher.encrypt (plaintext, trialciphertext);

                        // If the result equals the ciphertext, we
                        // found the key.
                        if (match (ciphertext, trialciphertext))
                           {
```

If a thread finds the correct key, it copies the key into a temporary byte array and writes the array reference into the shared `foundkey` variable. The JVM synchronizes multiple threads writing `foundkey`, preventing write-write conflicts.

```
                           byte[] key = new byte [32];
                           System.arraycopy (trialkey, 0, key, 0, 32);
                           foundkey = key;
                           }
                        }
                     }
                  });
               }
            });
```

At the conclusion of the parallel computation, the main program prints the results and exits as before.

```
      // Stop timing.
      long t2 = System.currentTimeMillis();

      // Print the key we found.
      System.out.println (Hex.toString (foundkey));
      System.out.println ((t2-t1) + " msec");
      }

   /**
    * Returns true if the two byte arrays match.
    */
   private static boolean match
      (byte[] a,
       byte[] b)
```

```
        {
        boolean matchsofar = true;
        int n = a.length;
        for (int i = 0; i < n; ++ i)
           {
           matchsofar = matchsofar && a[i] == b[i];
           }
        return matchsofar;
        }
     }
```

And that's it! To go from the sequential program to the SMP parallel program, we moved some of the variables' declarations to a different point to make them per-thread variables. We added a parallel team, parallel region, and parallel for loop. We moved the per-thread variable initialization into the parallel for loop's `start()` method. We moved the loop body into the parallel for loop's `run()` method. And we changed the code that saves the correct key to prevent thread conflicts. The parallel program's structure is much the same as the sequential program's, except the parallel computation code follows the nested parallel team–parallel region–parallel for loop idiom we studied in Chapter 6.

We will put off examining the FindKeySeq and FindKeySmp programs' running times until Chapter 8. Before moving on, though, let's consider a slight variation of these programs.

7.3 Early Loop Exit

As currently written, the AES key search programs try all possible values for the missing key bits, even though there's no need to continue once the correct key is found. Let's change that. In the sequential FindKeySeq program, when the correct key is found, it's easy enough to do an **early loop exit** by adding a break statement.

```
     for (int counter = 0; counter < maxcounter; ++ counter)
        {
        . . .
        if (match (ciphertext, trialciphertext))
           {
           . . .
           break;
           }
        }
```

However, if we add a similar break statement to the parallel FindKeySmp program, it won't work. The break statement will exit the loop in the parallel team thread that happens to find the correct key, but the other threads will stay in their loops trying useless keys. What we really want is for *all* the threads to exit their loops as soon as *any* thread finds the key.

Here's one way to do it. At the top of the loop, test `foundkey` and exit the loop if it is not null. This works because `foundkey` is initially null, and when the correct key is discovered, `foundkey` is set to a non-null array reference. Because `foundkey` is a *shared* variable (declared as a static field of the main program class), all the threads will be testing and setting the same variable, hence all the threads will exit their loops as soon as any thread finds the key. Reads and writes of an array reference variable are atomic, so no additional synchronization is needed when testing `foundkey`.

```
new ParallelTeam().execute (new ParallelRegion()
   {
   public void run() throws Exception
      {
      execute (0, maxcounter, new IntegerForLoop()
         {
         . . .
         public void run (int first, int last)
            {
            for (int counter = first;
                     counter <= last && foundkey == null;
                     ++ counter)
               {
               . . .
               if (match (ciphertext, trialciphertext))
                  {
                  byte[] key = new byte [32];
                  System.arraycopy (trialkey, 0, key, 0, 32);
                  foundkey = key;
                  }
               }
            }
         });
      }
   });
```

The FindKeySeq2 and FindKeySmp2 programs in the Parallel Java Library are sequential and SMP parallel versions of the AES key search program with an early loop exit as soon as the correct key is found. They will usually finish sooner than the original versions.

It's high time we measured and analyzed the SMP parallel AES partial key search program's performance. That will be the subject of Chapters 8, 9, and 10.