Parallel Key Search with Parallel Java

Chapter 7

Parallel code identification: The first step in designing the parallel version is to identify the computational code that will be executed in parallel.

For loop: It is the for loop in the middle of the main program. 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.

Variables

- **plaintext, ciphertext, partialkey, n**: They are used throughout the program. They will be shared variables.
- **keyLast4Bytes**: This holds the least significant 32 bits of the partial key. It is written by the main program and its value does not change afterward. 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 will be a shared variable.
- **foundkey**: This holds a copy of the key that was found to encrypt the plaintext correctly. It will be 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 easurements. They will remain local variables of the main program.

Synchronization

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 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.
- **keylast4Bytes**: 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.

Synchronization for foundkey variable

A separate array copy: 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.

Atomic reads/writes for array reference: 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.

Only for array reference: 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.

Atomic reads/writes:

Reads and writes of the following Java primitive types are guaranteed to be atomic:

- boolean,
- byte
- char,
- short,
- int,
- float
- object and array references

Nonatomic reads/writes

Reads and writes of the following types are not guaranteed to be atomic:

- long
- double
- 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. Ex: i++.

Synchronization needed when

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.

SMP Parallel Key Search Program

```
/**
 * original file: edu.rit.smp.keysearch.FindKeySmp2.java
 * @author  Alan Kaminsky
 * @version 05-Aug-2008
 */
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 FindKeySmp2 {

// Prevent construction.
    private FindKeySmp2() {
    }
// Shared variables.
    // 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 keyLast4Bytes;
    // The maximum value for the missing key bits counter.
    static int maxcounter;
    // The complete key.
    static byte[] foundkey;

    static boolean keyFound = false;

// Main program.
    /**
     * AES partial key search main program.
     */
    public static void main(String[] args)
            throws Exception {
        //Comm.init (args);

        // Start timing.
        long t1 = System.currentTimeMillis();

        // Parse command line arguments.
        // if (args.length != 4) usage();
//        plaintext = Hex.toByteArray(args[0]);
        plaintext = Hex.toByteArray("44656e656d65206d6573616ac4b10000");
//        ciphertext = Hex.toByteArray(args[1]);
        ciphertext = Hex.toByteArray("4d168e6cd45dcc264a3330e63fa6a52a");
//        partialkey = Hex.toByteArray(args[2]);
        partialkey =
Hex.toByteArray("b661ca5d5df7e4e66944751923247a91c1632bf1dc5821a5cd8d83fd4d
800000");
//        n = Integer.parseInt(args[3]);
        n = 20;

        // 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 program shared variables for doing trial encryptions.
        keyLast4Bytes =
                ((partialkey[28] & 0xFF) << 24)
                | ((partialkey[29] & 0xFF) << 16)
                | ((partialkey[30] & 0xFF) << 8)
                | ((partialkey[31] & 0xFF));
        maxcounter = (1 << n) - 1;

        // Do trial encryptions in parallel.
        new ParallelTeam().execute(new ParallelRegion() {

            public void run() throws Exception {
                int index = getThreadIndex();
                System.out.println("my index: "+index);

                execute(0, maxcounter, new IntegerForLoop() {
                    // Thread local variables.

                    byte[] trialkey;
                    byte[] trialciphertext;
                    AES256Cipher cipher;
//                    long m1, m2, m3, m4, m5, m6, m7;
//                    long m8, m9, ma, mb, mc, md, me;


                    // 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);
                    }

                    // Try every possible combination of low-order key
bits.
                    public void run(int first, int last) {
                        for (int counter = first; counter <= last &&
foundkey==null ; ++counter) {
                            // Fill in low-order key bits.
                            int last4Bytes = keyLast4Bytes | counter;
                            trialkey[28] = (byte) (last4Bytes >>> 24);
                            trialkey[29] = (byte) (last4Bytes >>> 16);
                            trialkey[30] = (byte) (last4Bytes >>> 8);
                            trialkey[31] = (byte) (last4Bytes);

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

                            // If the result equals the ciphertext, we
found the
                            // key.
                            if (match(ciphertext, trialciphertext)) {
                                byte[] key = new byte[32];
                                System.arraycopy(trialkey, 0, key, 0, 32);
                                foundkey = key;
                            }
                        }
                    }
                });
            }
        });

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

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

// Hidden operations.
    /**
     * 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;
    }

    /**
     * Print a usage message and exit.
     */
    private static void usage() {
        System.err.println("Usage: java [-Dpj.nt=<K>]
edu.rit.smp.keysearch.FindKeySmp <plaintext> <ciphertext> <partialkey>
```

```
<n>");
        System.err.println("<K> = Number of parallel threads");
        System.err.println("<plaintext> = Plaintext (128-bit hexadecimal
number)");
        System.err.println("<ciphertext> = Ciphertext (128-bit hexadecimal
number)");
        System.err.println("<partialkey> = Partial key (256-bit hexadecimal
number)");
        System.err.println("<n> = Number of key bits to search for");
        System.exit(1);
    }
}
```

Early Loop Exit

This program tries all possible keys: there's no need to continue once the correct key is found.

break works for sequential version: In the sequential program, when the correct key is found, it's easy enough to add break statement.

Similar solution does not work: 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.

All threads need to exit: What we really want is for all the threads to exit their loops as soon as any thread finds the key.

One solution: 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.

```
for (int counter=first; counter<=last && foundkey==null; ++counter)
```

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.