Parallel Programming with Parallel Java

Chapter 6

Parallel Team

A sequential program do some computation as following:

- Initial setup
- Execute the computation
- Clean up

An SMP parallel program, we looks like the following:

- Initial setup
- Create a parallel team
- Have the parallel team execute the computation
- Clean up

Parallel Team in Parallel Java

In Parallel Java, you get a parallel team by creating an instance of class ParallelTeam. The parallel team comprises a certain number of threads, which will be the ones to carry out the computation in parallel.

Three Choices for specifying the number of threads:

1. You can specify the number of threads at compile time as the constructor argument:

```
ParallelTeam team = new ParallelTeam(4);
```

2. You can specify it at run time by using the no-argument constructor.

```
ParallelTeam team = new ParallelTeam();
```

In this case, you specify the number of threads by defining the "pj.nt" Java system property when you run the program. You can do this by including the "-Dpj.nt" flag on the Java command line.

For example,

```
$ java -Dpj.nt=4 . . .
```

3. if you use the no-argument constructor and you do not define the "pj.nt" property, Parallel Java chooses the number of parallel team threads automatically to be the same as the number of processors on the computer where the program is running.

   (Parallel Java discovers the number of processors by calling the Runtime.availableProcessors() method, which is part of the standard Java platform.)

Normally, you will let Parallel Java choose the number of threads automatically at run time, so as to get the maximum possible degree of parallelism.

K Threads and a main thread

Once a parallel team has been created, K threads are in existence inside the parallel team object (Figure 6.1).

These are in addition to the Java program's main thread, the thread that is executing the main program class's main() method and that created the parallel team object.

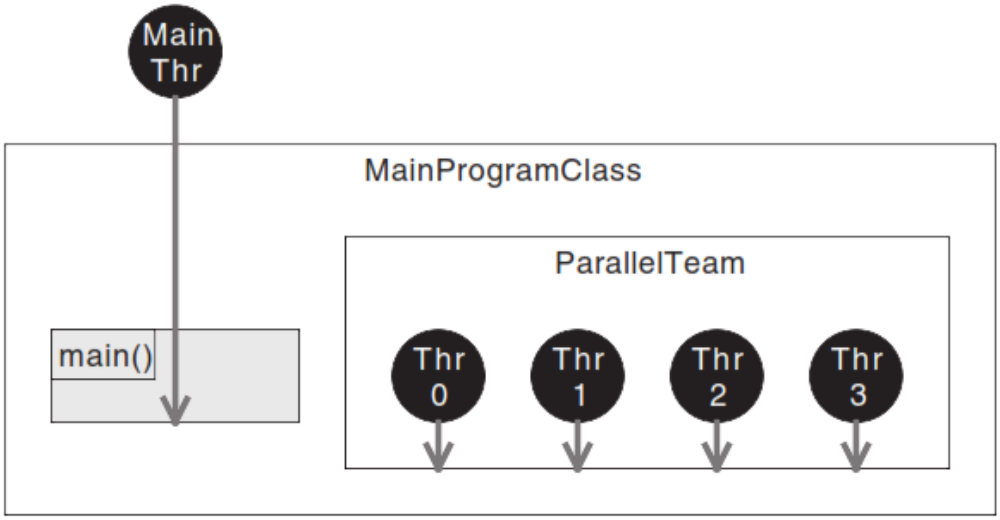

Figure 6.1 A parallel team witk $K = 4$ threads

Thread Index: Each parallel team thread has a thread index from 0 through K–1. Each thread is poised to execute the program's computation.

However, these threads are hidden, and you do not manipulate them directly in your program.

To get the threads to execute the program's computation, you use another Parallel Java class, namely class ParallelRegion.

Parallel Region

Parallel code: Whereas a parallel team contains the threads that will carry out the program's computation, a parallel region contains the actual code for the computation.

Class ParallelRegion is an abstract base class with three principal methods

- start()
- run()
- finish()

that you can override.

Writing the parallel code: To write the parallel portion of your program, define a subclass of class ParallelRegion and put the code for the parallel computation in the sub class's start(), run(), and finish() methods.

Create an instance of your ParallelRegion subclass and pass that instance to the parallel team's execute() method.

A convenient coding idiom is to use an anonymous inner class.

```
ParallelTeam team = new ParallelTeam(numbers.length);
team.execute(new ParallelRegion() {

   public void start() {
         // initialization code
        }

   public void run() {
         // parallel computation
        }

   public void finish() {
         // finalization code
        }
   });
```

Step by step parallel execution (Figure 6.2**)**

1. Each parallel team thread is initially blocked when they are constructed. They will not commence execution until signaled.
2. The main thread (executing the program's main() method) calls the parallel team's execute() method.
3. Inside the execute() method, the main thread calls the start method of parallel region.
4. After the start() method is done, the main thread signals each team thread to commence execution. The main thread then blocks.
5. All the team threads now proceed to call the parallel region's run() method. Here is where the parallel execution of the parallel region code happens.
6. When the run() methods return, each team thread signals the main thread. When all team members are done in run methods, and signalled the main method, all parallel threads cease to exist and the control of the execution is passed to the main thread.
7. When all threads are done in run methods, and signalled the main method, Main thread calls the finish method of the parallel region.
8. After the finish method, execute method of the Parallel Team object is finished.

9. Other codes after the execute method is executed in the program.

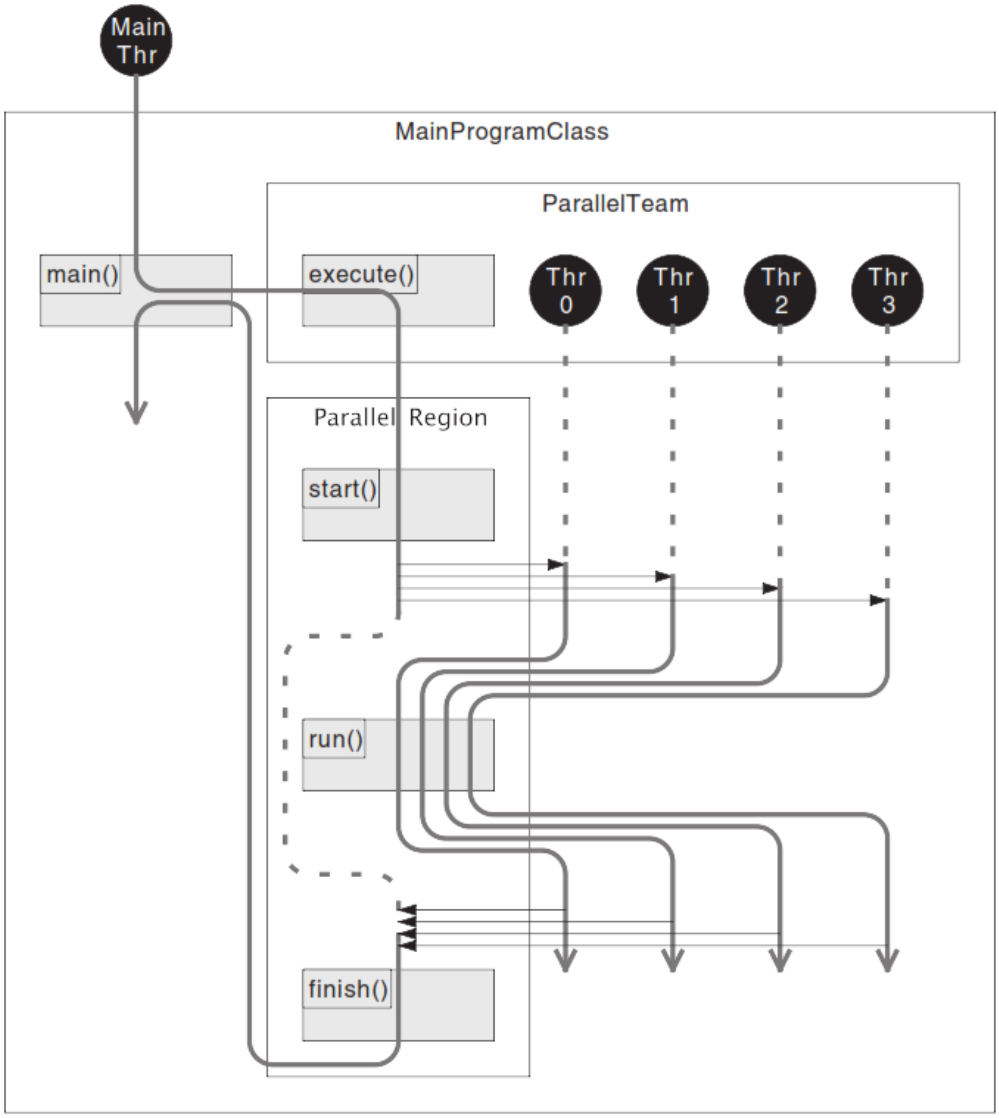

Figure 6.2 A parallel team executing a parallel region

In summary: Focusing on the parallel region's methods in Figure 6.2, the sequence of execution is the following:

- The start() method is executed by a single thread only once (main thread).
- The run() method is executed by K separate threads simultaneously.
- The finish() method is executed by a single thread only once (main thread).

```
/**
 * 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("b661ca5d5df7e4e66944751923247a91c1632bf1dc5821a5cd8d83fd4d800000");
//        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);

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

            public void start(){
                System.out.println("in start");
            }

            public void finish(){
                System.out.println("in finish");
            }
            public void run() throws Exception {

                int index = getThreadIndex();
                System.out.println("my index: "+index);

                AES256Cipher cipher;
                byte[] trialkey = new byte[32];
                System.arraycopy(partialkey, 0, trialkey, 0, 32);
                byte[] trialciphertext = new byte[16];
                cipher = new AES256Cipher(trialkey);

                int shareSize = maxcounter/getThreadCount();
                int first = shareSize*index;
                int last = shareSize*(index+1);
                    // Try every possible combination of low-order key bits.
                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;
```

```
                        keyFound = true;
                    }
                }
            }
        });

        // 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);
    }
}
```

Parallel For Loop (continue)

N iterations: Often, a program's computation consists of some number N of loop iterations.

Dividing N iterations to threads: To divide the N loop iterations among the K threads (processors),there is a helper class in Parallel Java: IntegerForLoop, which provides a parallel for loop.

IntegerForLoop Class: is an abstract base class with three principal methods—

- start(),
- run(),
- finish()

Implementing IntegerForLoop: To write the parallel for loop, define a subclass of class IntegerForLoop and put the loop code in the subclass's start(), run(), finish() methods.

Executing IntegerForLoop: in the parallel region's run() method, call the parallel region's execute() method, passing in the first loop index (inclusive), the last loop index (inclusive), and the IntegerForLoop subclass instance.

Anonymous inner class: Again, a convenient coding idiom is to use an anonymous inner class.

Example: Here is a parallel for loop with the index going from 0 to 99, inclusive.

```
ParallelTeam team = new ParallelTeam(numbers.length);
team.execute(new ParallelRegion() {

      public void run()
         {
         execute (0, 99, new IntegerForLoop()
            {
            public void start()
               {
               // Per-thread pre-loop initialization code
               }
            public void run (int first, int last)
               {
               // Loop code
               }
            public void finish()
               {
               // Per-thread post-loop finalization code
               }
            });
         }
      });
```

Here's what happens inside the parallel team (Figure 6.3).

The parallel team threads are executing the parallel region's run() method simultaneously. Each team thread executes the following statement:

execute (0, 99, new IntegerForLoop()...);

Each thread, therefore, first creates its own new instance of the IntegerForLoop subclass.

Each thread then calls the ParallelRegion's execute() method, passing in the loop index lower bound (0), the loop index upper bound (99), and the thread's own IntegerForLoop object.

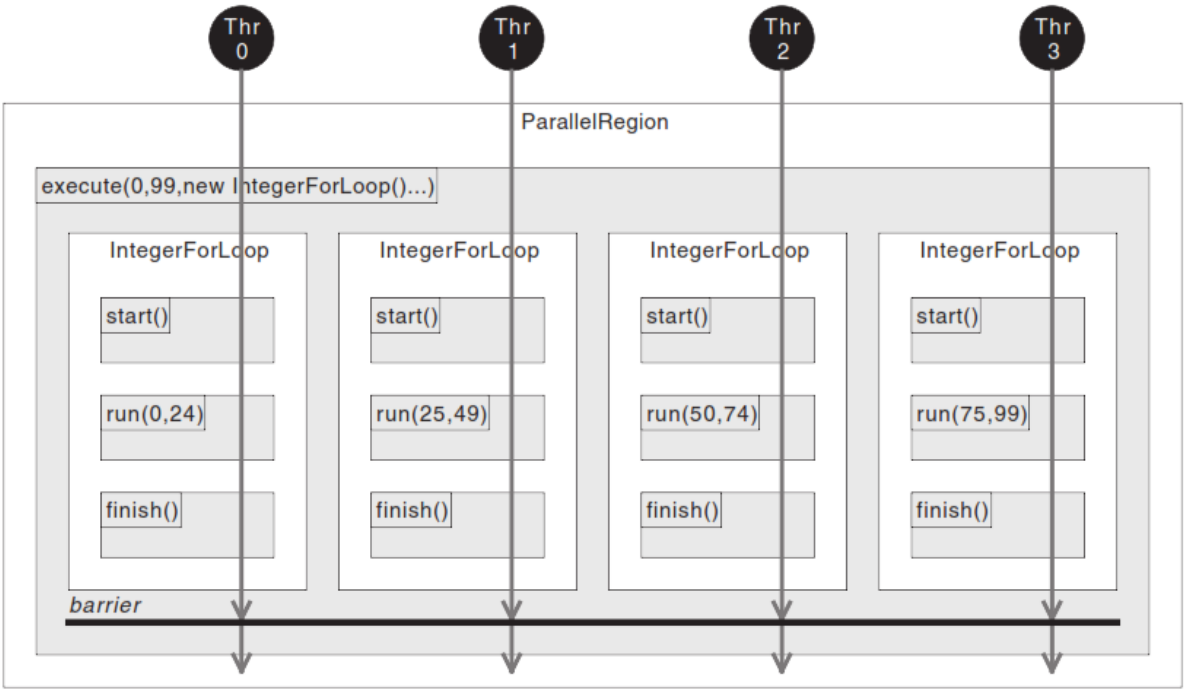

Figure 6.3 A parallel team executing a parallel for loop

Partitioning the range: ParallelRegion's execute() method partitions the complete index range, 0–99, into K equal-sized subranges, or chunks, namely 0–24, 25–49, 50–74, and 75–99.

Loops for Each Thread: Each thread now proceeds to call its own IntegerForLoop object's start(), run(), and finish() methods in sequence.

Thread run methods: Thread 0 passes the arguments first = 0 and last = 24, thread 1 passes the arguments 25 and 49, thread 2 passes the arguments 50 and 74, and thread 3 passes the arguments 75 and 99.

Waiting at the Barrier: After completing this sequence of execution each thread waits at a barrier.

Barrier: A barrier is a thread synchronization mechanism that ensures that none of the threads will proceed past the barrier until all the threads have arrived at the barrier.

When all done: When the last thread finishes its portion of the parallel for loop and arrives at the barrier, the barrier opens.

Finishing execute method: Each thread resumes execution, returns from the ParallelRegion's execute() method, and continues executing the code that comes after the parallel for loop in the parallel region's run() method.

start and finish methods: If no initialization or finalization code is necessary, simply omit the start() or finish() method or both.

Parallel For Loop run method

ParallelForLoop's run() method's job is to execute the loop iterations for the chunk whose first and last loop indexes, inclusive, are passed in as arguments.

Thus, the run() method typically looks like this.

```
   public void run (int first, int last)
      {
      for (int i = first; i <= last; ++ i)
         {
         // Code for loop iteration i
         }
      }
```

Variables

where should we put the variables for a Parallel Java program.

Following the aforementioned coding idioms gives rise to a nested class structure (Figure 6.4).

The parallel for loop class is nested inside the parallel region class, which in turn is nested inside the main program class.

```
public class MainProgramClass
   {
   // Shared variable declarations
   static int a;

   public static void main(String[] args) throws Exception{

      // Main program local variable declarations
```

```
      int b;
      new ParallelTeam().execute (new ParallelRegion()
         {
         public void run()
             {
             execute (0, 99, new IntegerForLoop()
                {
                // Per-thread variable declarations
                int c;
                public void run (int first, int last)
                   {
                   // Loop local variable declarations
                   int d;
                   for (int i = first; i <= last; ++ i)
                      {
                      // Code for loop iteration i
                      }
                   }
                });
             }
         });
      }
   }
```

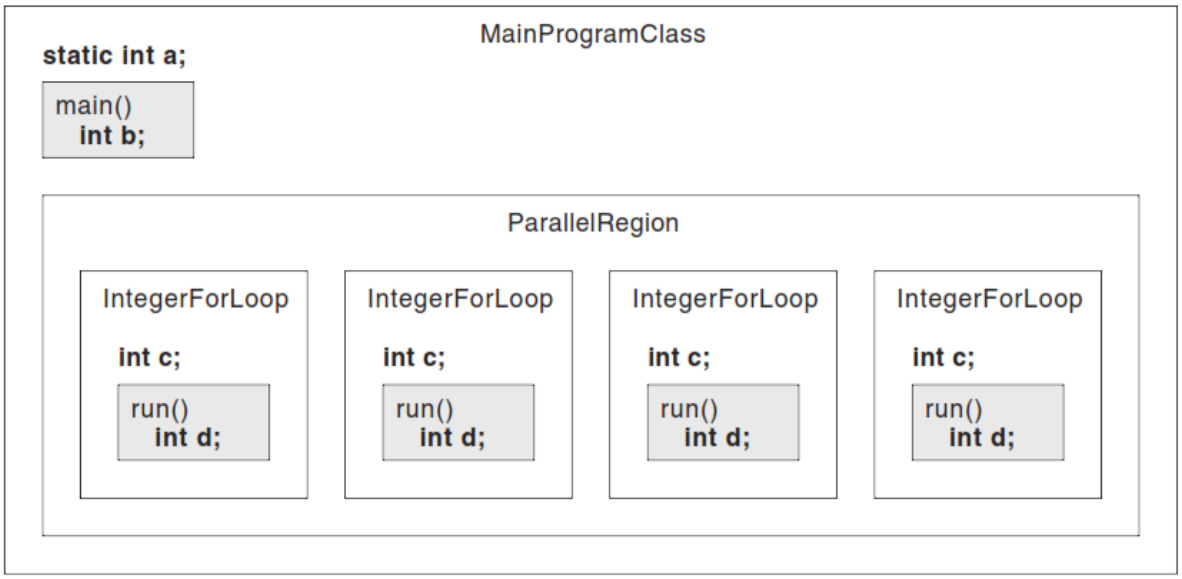

Figure 6.4 Variable declarations in a Parallel Java program

Shared variables (static variables)

They are declared as static fields of the main program class. Ex: int a.

There is only one instance of each shared variable.

The main program thread and all the parallel team threads can access every shared variable.

If one thread changes the value of a shared variable, all the other threads will see that new value.

Per-thread variables

They are declared as instance fields of the IntegerForLoop subclass. Ex: int c;

Such variables can be accessed by code anywhere in the parallel for loop subclass.

Each thread gets its own separate instances of the per-thread variables.

If one thread changes the value of a per-thread variable, this will not affect the value of the corresponding per-thread variable in any other thread; per-thread variables are not shared.

The parallel for loop's start() and finish() methods can be used to initialize and finalize these perthread variables.

Loop local variables

They are declared as local variables of the parallel for loop subclass's run() method. Ex: int d.

The loop control variable (such as i) is also a loop local variable. Loop local variables can be accessed only by code in the parallel for loop's run() method, and each thread gets its own separate instance of each loop local variable.

Main program local variables

They are declared as local variables of the static main() method.

Such variables are used only by the main thread executing the main() method.

Code inside the parallel region or parallel for loop cannot access main program local variables.

Taking care of shared variables

When a variable is shared, we must make sure that the threads do not conflict with each other when they access the shared variable.

A thread can do certain operations on a variable:

- **reading a variable**; that is, the thread can examine the variable's state without changing its state.
- **Writing a variable**; that is, the thread can change the variable's state.

- **Updating a variable**; that is, the thread can read the variable's state, compute a new state based on the old state, and write the new state back into the variable. For ex: i++

A conflict can arise when two or more threads do certain operations on a variable at the same time:

- **read-write conflict:** can arise if one thread reads a variable at the same time as another thread writes or updates the variable; the reading thread may read an inconsistent state where part of the state is the old state and part of the state is the new state.
- **Write-write conflict** can arise if two threads write or update a variable at the same time; one thread's writes may wipe out some of the other thread's writes, again leading to an inconsistent state.
- **Two concurrent reads:** There is no conflict if two threads read a variable at the same time.

Synchronization

if no conflict: If multiple threads cannot conflict when they access a shared variable, then we don't have to do anything special.

if they conflict: But if multiple threads can conflict when they access a shared variable, we must synchronize the threads to eliminate the potential conflict.

Synchronization PJ: Parallel Java has several constructs for thread synchronization that we will study later.

Synchronization adds overhead to the parallel program and can significantly increase the program's running time.

Careful Analysis: We must carefully analyze each shared variable for thread conflicts and introduce thread synchronization only where it is absolutely needed.