Lab 8

In this lab, we’ll use a framework for efficiently implementing parallel reductions in Java.

Implementing reductions using Java’s ForkJoin framework

Yesterday, we talked about parallel reductions where different processors handle different parts of a problem. In Java, the way to use multiple processors is to create multiple threads, which live in the same address space (so they can share variables etc) but which each have their own call stacks and program counters (so they can run different instructions and store their own copies of local variables). We’ll be using a specific library that provides particularly lightweight threads, ones that are cheap to create, destroy, and switch between. This allows us to create many more threads than there are processors, making the code independent of the number of processors available (which can vary even on a single system depending on what else is running).

The directory /home/courses/cs180i/lab8 contains an example of this library in action. The file jsr166.jar is the library itself and FJExample.java is code using it. Copy these files. To compile and run them from the command line, you need to add an extra flag:

```
javac -Xbootclasspath/p:jsr166.jar FJExample.java
java -Xbootclasspath/p:jsr166.jar FJExample
```

To use it in Eclipse, import jsr166.jar into your project (“Import... File system”, making sure “Create selected folders only” is selected) and then right-click on it to choose “Add to Build Path”. You also need to go into “Run -¿ Run configurations”, select FJExample as the main class, and (under “Arguments”) add -Xbootclasspath/p:jsr166.jar in the “VM arguments” box.

When you’re able to run this program, you’ll get output 15, which is the sum of an array containing the numbers 1 through 5. Let’s look through the code to understand how this program works. The bulk of the work is done by the SumArray class, which extends the library class RecursiveTask. (The generic parameter Integer here means that the task returns an Integer.) It has instance variables to store the problem to be solved and a compute method that does the actual calculation.

For small problem instances (those with size less than SEQUENTIAL_THRESHOLD), compute uses a serial algorithm to avoid the overhead of creating more threads. For larger instances, it creates the two subproblems. One of these is solved using a direct call to compute, but the other uses compute called indirectly via fork. A call to fork starts the task in a new thread. The new thread calls compute, but the call to fork returns as soon as the new thread is created rather than waiting for its completion. This allows the two subproblems to be solved at the same time; one by the original thread and one by the newly created one. Whenever fork is used, a later call to join is necessary. Calling join causes the original thread to wait until the new thread finishes; join returns the result of the computation done by the new thread.

The other aspect of the program is setting up the call to the recursive part. The compute and fork methods should not be called except from within compute. The initial call from outside compute must be made by invoke, a method of ForkJoinPool that takes a RecursiveTask. The invoke method starts the first thread in order to get the recursion started. ForkJoinPool is a set of Java threads on which the libraries lightweight threads run. The Java threads are heavy, meaning it is more expensive to create, destroy, and switch between them. To avoid paying these costs for each lightweight thread, the heavy threads are created once at the beginning of the program and stored in the pool. Then they are given lightweight threads to run when possible, but they are returned to the pool rather than being destroyed when a lightweight thread

---

1Borrows heavily from Dan Grossman’s notes on ForkJoin included as part of http://www.cs.washington.edu/homes/djg/teachingMaterials/spac/.
finishes. Thus, they are “recycled” by all the lightweight threads running during the program, saving most of the overhead.

Hopefully that clarifies how the program works. Please ask if you have further questions— I know that there are both new concepts and new classes appearing in the sample code.

Once you are comfortable with the code, it is time to begin modifying it. Start by creating a larger array to sum. Since the current size is less than SEQUENTIAL_THRESHOLD, the program currently doesn’t use any parallelism. Fix this by using a larger array.

Next, change the program so that it does other reductions. What needs to change for the program to return the value of the largest element (max)?

Once you’ve done that, create a version of indexOf that returns the first index where a particular value appears or -1 if that value is not in the array.

Next, modify the program to find the second largest value as we discussed in class. For this, you’ll need to modify the return type of compute, which requires changing the parameter of the RecursiveTask class being overloaded.

If you have additional time, try to think up and implement other reductions.