Timing Quicksort in Parallel

Lab 9

Wednesday, April 3, 2013

CSC 062: Spring, 2013

In this lab, we'll once again be playing with sorting algorithms, though this time to make them more efficient by using parallelism. As usual, you may work in pairs on this lab.

Getting started

Create a new project in `Eclipse` and then go to `Terminal` and copy over the two ".java" files from `/common/cs/cs062/labs/lab9`. Load the Java files into your project.

You will notice that this version of Quicksort is a bit clunkier than earlier versions you have seen. It is invoked by creating a new QSManager and then invoking its `run` method. Similarly each recursive call creates a new QSManager and then calls its run method. The reason for the extra overhead of creating new objects for each call is to make it easier to generalize this for parallel execution.

Start by running the main method of QSManager to get timing information on quicksort. Notice that the code runs the sort routine 10 times to "warm up" the code, and then runs it another 10 times to get timing values. It reports each of those times as well as the minimum of those 10 times.

Answer these questions:

1. Why is there variance in these numbers? (Hint, it is more than just the application continuing to warm up.)

2. Why does it make sense to take the minimum value?

The program also prints out the first 10 elements of the sorted array so that you can make sure the array is correctly sorted after you make modifications to the code later in the lab.

Write down the minimum time for 10000 elements and 20000 elements by changing the value of the constant `NUM_ELEMENTS`.

3. Do these numbers make sense given our analysis of the big-O complexity of quicksort?

Make it run in Parallel

Modify the code in `QSManager` so that the recursive calls run in parallel. This can be accomplished by making `QSManager` extend `Thread` and invoking it with `start()` rather than `run` when you want to spin off a separate thread. We would like this to run as efficiently as possible, so only create a single new thread when you make the recursive calls (and the initial call can also run in the same thread as the rest of the main program). This code should be very similar to that of the second example on the code page for Lecture 20, but will only involve changing a few lines of the existing code.

Don't forget to wait for the new thread to complete before returning from the `run` method. Also, be careful of the order that you write the code to ensure that it really runs in parallel and not sequentially.

Have a TA or instructor come over and check your program before you record the times, below.

Using this version of the program, write down the minimum times for 10000 and 20000 elements in the array.

4. Explain why you think this version of QuickSort is faster (or slower, depending on your results) than the previous version.

Using the ForkJoin Framework

Now that you have it running in parallel, make it even faster using the ForkJoin framework from Java 7. This version should be similar to the first code examples associated with lecture 22, except that your class will extend `RecursiveAction` rather than `RecursiveTask` (because it is not a function).

Make sure that `QSManager` class imports the appropriate classes from `java.util.concurrent`.

Using this version of the program, write down the minimum times for 10000 and 20000 elements in the array. Also, answer the following question.

5. Explain why you think this version of QuickSort is faster (or slower, depending on your results) than the previous versions.

What to turn in

Save your answers to the questions above and the times for the three different versions of the program for 10,000 and 20,000 elements in a file name "Lastname-Lab9.txt," or "Lastname1-Lastname2-Lab9.txt" if you worked with a partner, where you replace *Lastname* with your last name. Include at the end of the file the code for your last version of the program.