Computer Science 62
Lab 9

Wednesday, April 1, 2015

Timing Quicksort in Parallel

In this lab, we'll once again be playing with sorting algorithms! This time our goal is to make them more efficient by using parallelism. You may work in pairs on this lab.

Getting started

The starter files for this lab can be found in `/common/cs/cs062/labs/lab09`. 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.

Please answer these questions. You can open up a text editor and write your answers there if you wish.

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

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 10,000 elements and 20,000 elements by changing the value of the constant `NUM_ELEMENTS`.

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

Running 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 start a separate thread. (Refer to the "Parallelism and Concurrency" text or your lecture notes for additional details).

We would like the code 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). Your code should be very similar to that of our final attempt at summing an array using Java's `Thread` class (see Lecture 23). 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. Using this version of the program, write down the minimum times for 10,000 and 20,000 elements in the array.

3. 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 code examples from lecture 23 except that your class will extend `RecursiveAction` rather than `RecursiveTask` (because `compute` needs no return value). Make sure that `QSManager` imports the appropriate classes from `java.util.concurrent`.

Using this version of the program, write down the minimum times for 10,000 and 20,000 elements in the array. Also, answer the following question.

4. 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 text file named "Lab09_LastNameFirstName". *Include at the end of the file the code for your last version of the program.* If you worked with a partner, put both of your names at the top of the text file.