# Parallelism

Lab 8

Wednesday, November 1, 2017

CSC 062: Fall, 2017

## Introduction

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.

We'll also be using `git` a tiny bit more in this lab.

## Getting Started

The starter files for this lab can be found in `/common/cs/cs062/labs/lab08`. However, instead of copying them like you usually do, we'll use `git` to create a synchronized clone of them. Start by opening up a terminal and `cd`ing into your workspace directory. Then enter the following command to create a clone of the starter code:

```
git clone /common/cs/cs062/labs/lab08
```

It should say:

```
Cloning into 'lab08'...
done.
```

As we saw in the last lab, `git` is a tool that keeps track of different versions of code and allows you to combine versions from multiple sources. It also allows you to roll back to a previous version if you find out that you made a mistake.

By cloning a repository, you have just created a copy of the starter code, along with all of its history of changes, in such a way that you can now add your own changes and then eventually send them back to the origin repository.

Let's look at the history of the starter code by typing `git log`. This will show you a record of each commit, which is a set of changes that someone added to the repository. When you do this, you'll notice that something isn't quite right—the most recent commit says something about introducing errors!

Luckily, with `git` we can always go back in time if we made a mistake. To do this, type:

```
git checkout 466d560
```

This will temporarily update your code to a specific version (The number `466d560` was the (unique) start of one of the commit numbers from the `git log` output (and you could have tab-completed it by typing `46<TAB>`)). By doing this, you can look at a specific version of the code, but if you type `git status`, you'll see that it says `''HEAD detached.''` This rather troubling state of affairs happens when the current version you're looking at (called `HEAD`) isn't the most recent version on a branch. When you make changes, it's best to add them to the end of a branch, so that things don't get too confused.

To list the current branches, use `git branch`. You'll see that there's a branch called `error`, which you're currently on, and another branch called `master`, which is the default branch in git. So let's see what's in the `master` branch: run `git checkout master` and then `git log`. Now you should see a commit that *reverts* the errors introduced in the error branch. Effectively, the `error` branch is a dead-end that's now out-of-date. Note, if we wanted to see what the errors were, we could run `git diff 1f2ce3c 5ef789b` (or just `git diff master error`) to have `git` print out the exact differences between the fixed version and the error version. If we wanted to convince ourselves that the error really was reverted, we could run `git diff master 466d560` and hope that it prints out nothing — no changes between the current `master` branch and the original commit `466d560`.

This is as far as we'll go with `git` this lab. At this point, we need to import this code into Eclipse. The best way to do this is actually to start a "New Java Project" and uncheck the "Use default location" box, telling it instead to use the code that we just got using `git`.

## The Quicksort Code

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.

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.

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.)

2. Write down the minimum time for 10,000 elements and 20,000 elements by changing the value of the constant `NUM_ELEMENTS`. 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`. 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. 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` Environment

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 lectures 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 Hand 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 `answers.txt.` Export your Eclipse code as usual and submit `answers.txt` along with your `bin/` and `src/` directories as usual. Don't forget to put both your name and your partner's name in the `.json` file.