CSC2002 Assignment 1
Cameron Broomfield, BRMCAM003

Introduction

The aim of this project is to study the difference in performance of a sequential program compared to a parallelized version of the program and to try to ascertain the optimal sequential cutoff point for this particular problem. The program with which we are testing our sequential/parallel performance comparison is Terrain Classification. This is a mapping-type program where, we are supplied with a grid of terrain height values and we must classify each data point as either TRUE (it is a basin) or FALSE (it is not a basin). Basin classification is based on the 8 data points surrounding the current point.

The exact same terrain classification algorithm was implemented as a sequential program (in SerialTest.java) as well as a parallel program (in ForkJoinTest.java). The latter used the Java Fork/Join framework as the method of parallelization.

Through correct implementation, I expect to see a significant speedup in performance in the parallel implementation when compared to the sequential implementation.

Methods

Step 1:

My first objective was to implement the terrain classification algorithm sequentially and ensure I am receiving the correct output. After a few hours of fighting with floating point values, I eventually got the correct output (tested using the given input files small_in.txt, med_in.txt and large_in.txt)

Step 2:

Implement the algorithm within the Fork/Join framework. After implementing the parallelization part of the ForkJoinTest.java class, I copied the terrain classification algorithm from the SerialTest.java class. The lines 14 - 38 in SerialTest.java are exactly the same as the lines 27 - 52 in ForkJoinTest.java. Because I was using a 2D-array to store the grid of data points, splitting both the rows and columns into high and low values for each thread proved tricky. Instead I split the array by rows only.

Step 3:

Test the parallel program with various sequential cutoff points to determine which value would provide the best performance. Using the med_in.txt, large_in.txt and 2*large_in.txt files as input, I tested performance with sequential cutoff values of 2, 10, 50, 100, 200, 500, 1000 and 2000 (see Fig. 1-3 under Results section).

Note: there was not much of a difference in performance when using different cutoff points. I opted to compare the sequential performance with the parallel performance with three different sequential cutoffs: 10, 100 and 1000, to get a more complete picture.

Step 4:

Test the performance of the sequential and parallel program using different sized inputs. The supplied input files had the following sizes: small_in.txt = 256 × 256, med_in.txt = 512 × 512 and large_in.txt = 1024 × 1024. I was, however, unsatisfied with these data sizes. I wanted to compare the two programs using a larger data size to accentuate the difference in performance.

In line 9 of Main.java, I declare the 'coef' (short for coefficient) variable. This is used to artificially increase the data size by multiplying the data size by 'coef'. 'coef' is equal to 1 by default, meaning if you input a data size of 1024 × 1024, the 2D float array called 'data' will be 1024 × 1024. However, if for example 'coef' = 2, the 'data' will be a 2048 × 2048 sized array. This effectively copies the large_in.txt input file twice into the new, larger array (i.e. the values data[0][0] to data[1023][1023] are repeated again in values data[1024][1024] to data[2047][2047]). Although the data input has simply been repeated to increase the data size, the algorithm doesn't know this and is blind to the fact that it is about to run through the same data as before. This was a simple way to increase the size of the input without having to create my own input files from scratch. Using this 'coef' variable, I tested the performance with an additional two data sizes: 2×large_in.txt = 2048 × 2048 and 4×large_in.txt = 4096×4096. Using these larger inputs, I was much happier with the difference in performance between sequential and parallel implementations.

Step 5:

Record and analyse the performance. The performance was measured using 5 different input sizes (256, 512, 1024, 2048, 4096) and 3 different sequential cutoff values (10, 100, 1000). For each input size and cutoff point, I ran the test 20 times and took the average. There was very little variability in the recorded times, most of them within a few milliseconds of the average, so I was happy to leave the number of runs at 20. The data was then plotted and analysed in R.

Problems:

The first problem I ran into was an issue with data types. I was getting a mostly correct output, but the number of basins that I reported was a few more or a few less than the example output supplied to us. For e.g. the med_in.txt input was giving me 133 basins instead of 130. I took one of the incorrect data points in my output and calculated by hand the difference in height of the surrounding values. I noticed one of the values was exactly 0.01. After speaking to a few people who had exactly the same output, we figured out that it was because we were using double data types and should instead use float. This was an interesting (and frustrating) encounter with floating point imprecision.

The second problem was that my parallel times were significantly longer than my sequential times. They were consistently about twice as slow as the sequential program. I then

remembered that my virtual machine only had access to two of the four cores my architecture has. I shut down the machine, changed the settings to four cores and booted back up, but alas now my parallel time was four times as slow! I decided to remove the virtual layer of hardware and perform the time trials on the host OS. Running the programs through the JGrasp IDE on the host OS gave me extremely consistent times, and best of all, the parallel program was now faster than the sequential program (for certain dataset sizes).

Results and discussion

Experiment 1: Cutoff vs time

This experiment ran three tests, studying the relationship between sequential cutoff point and the time the parallel program takes. Each of the three tests used a different dataset size. This was done to try to ascertain the optimal cutoff point relative to dataset size.

Test 1: Dataset size = 512

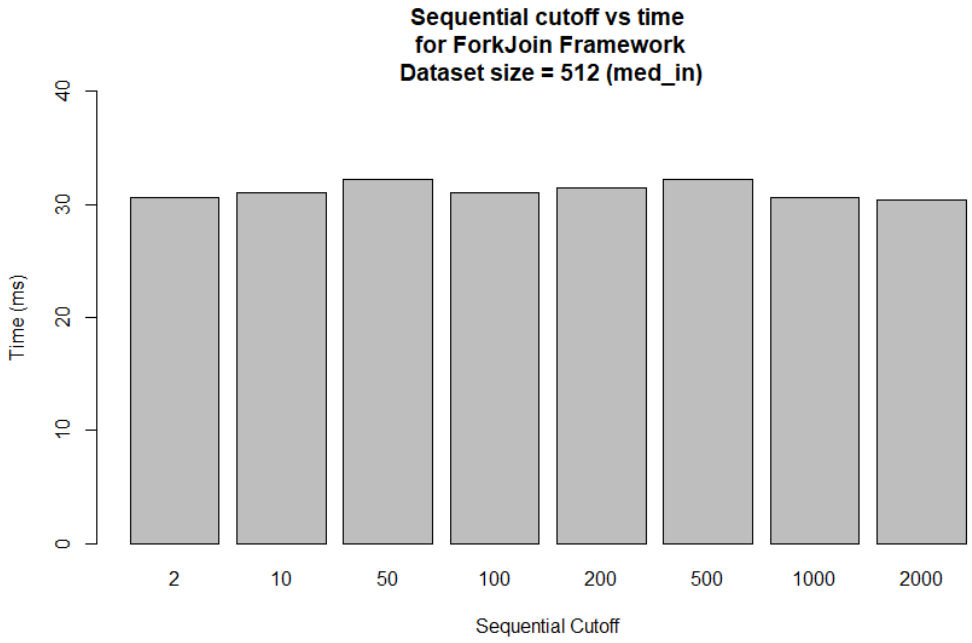

Figure 1, Dataset size = 512

In this test, there is no significant difference between any of the cutoffs. It was later noticed in experiment 2 that the difference in performance between the sequential and parallel programs was not significant at a dataset size of 512 (see Figures 4 - 6) and this is likely responsible for the absence of a noticeable difference.

Test 2: Dataset size = 1024

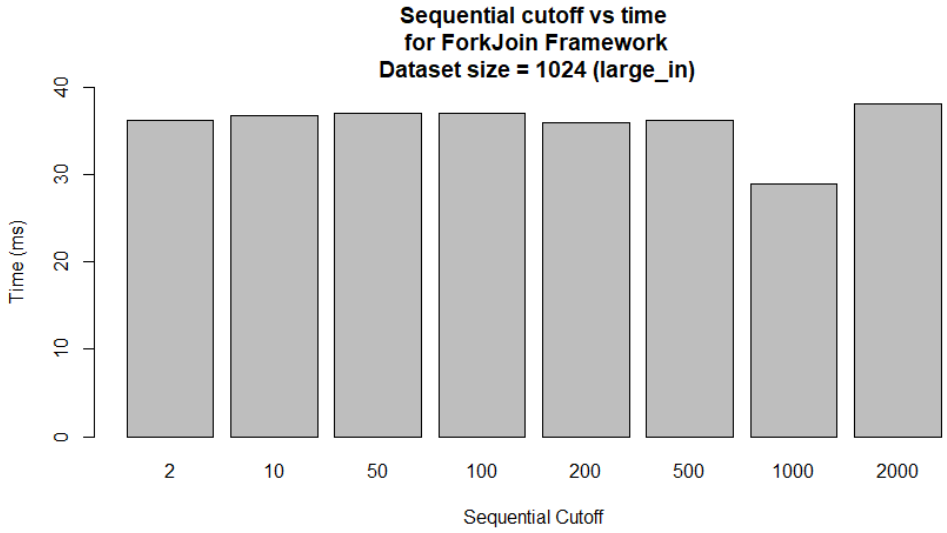

Figure 2, Dataset size = 1024

From this test, there are no clear losers but there seemed to be a clear winner. A cutoff of 1000 is unique in the fact that it is just slightly less than the dataset size, meaning that the dataset is split only once into two separate threads.

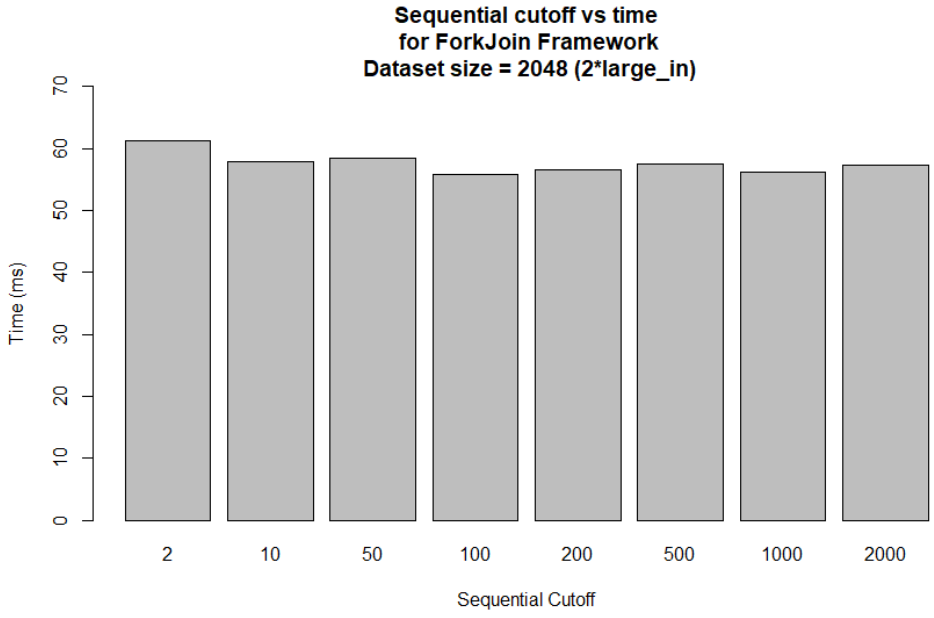

Figure 3, Dataset size = 2048

There were no clear winners in this test but the losers seem to be the lower cutoff points. This is likely the point at which there are too many threads and the returns of parallelization are diminishing. I would have assumed that a cutoff of 2000 would be a clear winner just as

1000 cutoff point was a clear winner in the previous test (Fig. 2), however this was not the case.

The results from experiment 1 were quite inconclusive. I was hoping to see a direct relationship between sequential cutoff point and dataset size, but this was not apparent.

Experiment 2: Dataset size vs time

The aim of this experiment is to compare the relative speedup in performance between a sequential program and the parallel version of the same program. This experiment ran three tests comparing sequential to parallel performance using different dataset sizes. Because experiment 1 was inconclusive, I chose cutoff values of 10, 100 and 1000. Each of the three tests used the same data sizes but had their respective sequential cutoff points.

Test 1: cutoff = 10

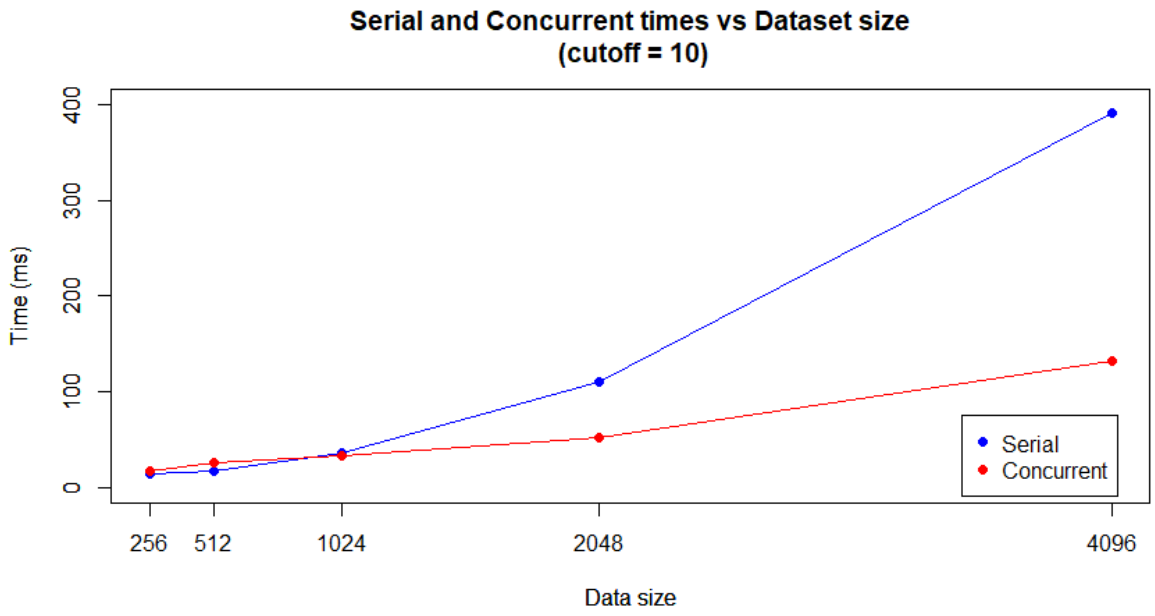

Figure 4, cutoff = 10

In this test, the parallel program showed worse performance when compared to the serial program for dataset sizes 256 and 512. Although not terribly worse, it was worse nonetheless. Parallel was only marginally better than serial at dataset size 1024, however, every dataset size thereafter, the parallel program showed greatly better performance.

Test 2: cutoff = 100

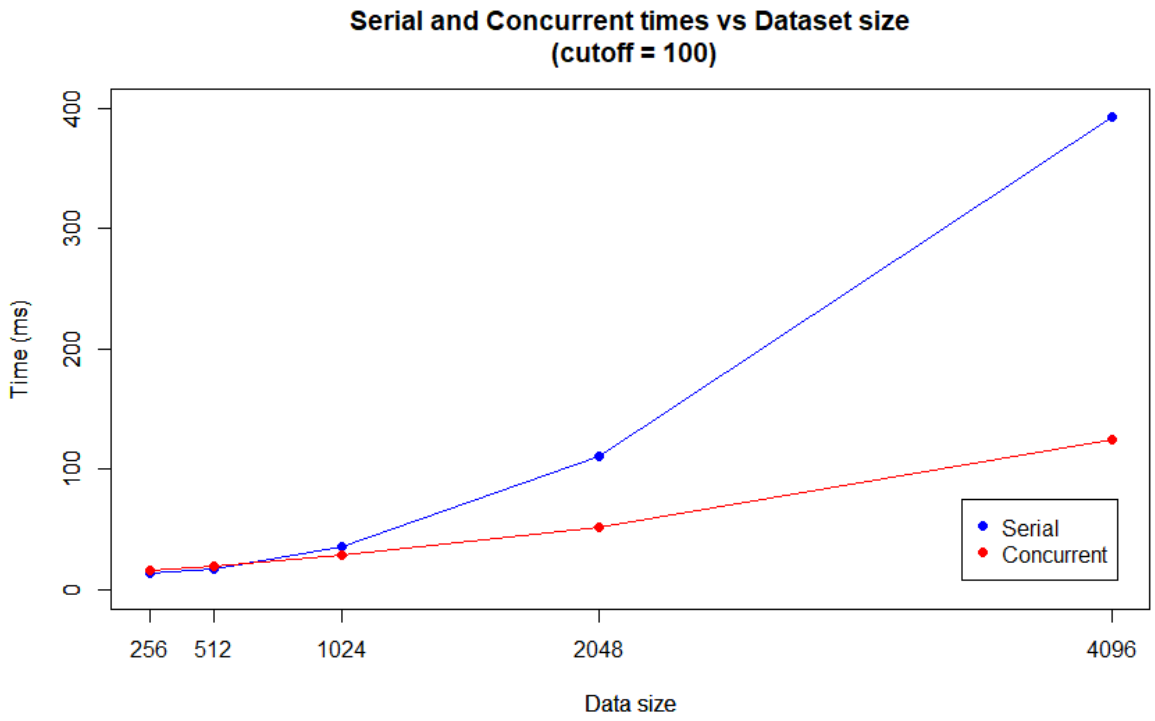

Figure 5, cutoff = 100

At data sizes 256 and 512, the serial and parallel programs had roughly equal timings, although parallel was always the worse of the two. At data size 1024, the parallel program showed slightly better performance than the serial program, comparatively better than in test 1 though. At data size 2048 and 4096, the parallel program showed similarly improved performance compared to test 1.

Test 3: cutoff = 1000

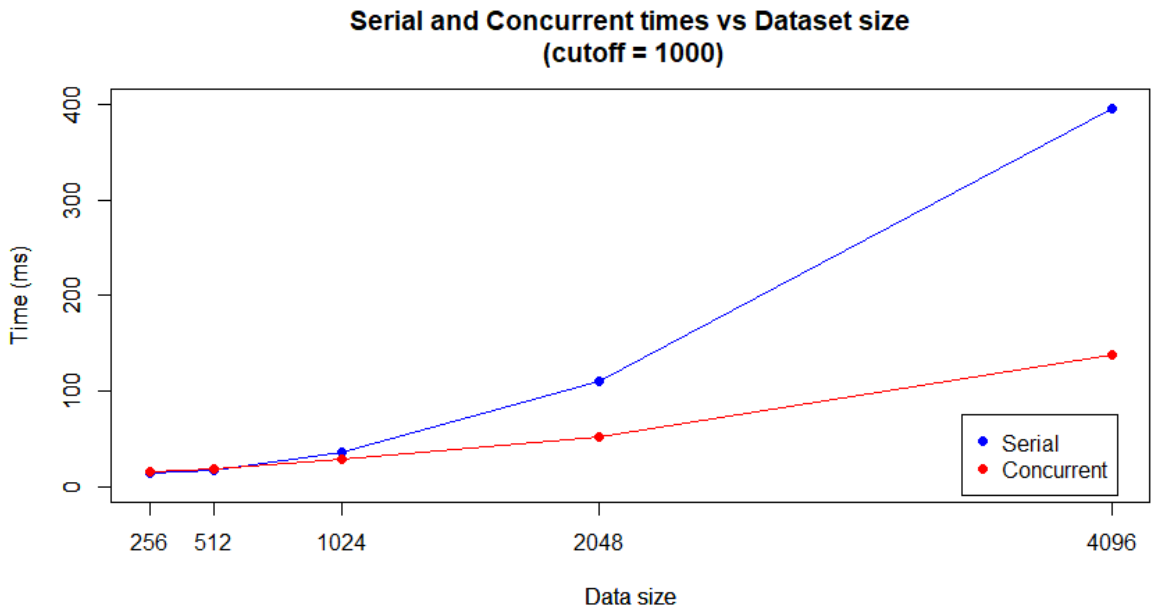

Figure 6, cutoff = 1000

Data sizes 256 and 512 exhibited the exact same performance in both the serial and concurrent. This is because both of those data sizes are smaller than the sequential cutoff. This effectively makes the parallel program behave as a sequential program for data sizes that small, thus giving the same performance as the sequential program. Across all three tests, the cutoff value did not seem to affect performance at the larger dataset sizes.

Conclusions

- What is an optimal sequential cutoff for this problem? What is the optimal number of threads on your architecture?
  The goal of experiment 1 was to answer this question. Experiment 1 was inconclusive. I was hoping to find a relationship between dataset size and cutoff point but was unable to. For the most part, all cutoff points performed similarly regardless of data size. The only significant difference seen was at cutoff = 1000 in Figure 2 with data size of 1024. I was hoping to see a similar difference at cutoff = 2000 in Figure 3 with data size of 2048, but I did not. Because testing time against cutoff point was inconclusive, it is difficult to draw a conclusion about the optimal number of threads on my 4-core architecture.

- Is it worth using parallelization (multithreading) to tackle this problem in Java? For what range of data set sizes does your parallel program perform well?
  Yes, it is worth using multithreading for this problem. For dataset sizes 1024 and larger, the parallel program consistently performs better than the sequential program, yielding greater and greater returns the larger the data size gets. Below 1024 however, the parallel program performs slightly worse than the sequential program.
- What is the maximum speedup obtainable with your parallel approach?
  For a data size of 2048, the parallel program was about 2.2 times as fast. For the data size of 4096 the parallel program was 3.2 times as fast. For any data size smaller than that, the speedup was relatively insignificant.