Introduction

Aim

The primary aim of this assignment is to gain an understanding of multi-threaded parallelism by comparing the efficiency of a sequential (non-parallel) program with that of a parallel multi-threaded one. Both programs need to process and classify points on a graph as either basins or non-basins and return a list of basin coordinates as well as the number of basins in the given terrain. To measure the efficiency, I conducted several tests performing these classifications on terrains with vastly varying sizes. The experiment will compare the time taken to classify basins sequentially and using parallelism (speedup).Furthermore, this assignment aims to find the optimal number of threads/sequential cutoff (i.e the optimal data size to execute sequentially in a parallel program) in relation to the data size. To get clearer understanding of this optimal value, I will compare the execution times on a machine with two cores, and another with eight cores.

Hypothesis

I think that the parallel program will perform significantly better through all the data sets. I also think that the larger the data size is, the more noticeable the differences in the speedup.

Methods

Object-Orientated Program Design

Before a detailed discussion of the experiment results can be presented, it is important to first discuss the Object-Orientated Programing design of the program. This is because the design of the program can greatly affect the performance. Badly designed programs produce bad inefficient results. Due to the nature of the designated problem, my sequential and parallel programs have slightly different class structures.

Sequential Program:

The sequential implementation has only one class titled:

```
class SequentialBasins
```

This class contains all the necessary methods to solve this problem. The first step is to read our data from file and store it in an appropriate data structure. To achieve this, I used a method titled:

```
static float[][] read_from_file()
```

which takes no parameters and returns a 2-dimensional array that will represent the points on the terrain. Within this method I have a method that will find the size (number of rows and columns) of my two-dimensional array. It is worth mentioning that the names of the input and output files are stored as global variables and are the first two arguments of the java program.

After reading from file the timer is started using the tick() method. Another method called:

```
processMatrix(float [][] grid)
```

This method takes in a two-dimensional array and sequentially performs the check on each point on weather it is a basin or not. The check starts from the second line second column and ends at the penultimate line and penultimate column, to avoid processing edges of the terrain (which the problem specifies can never be basins. See fig 1). Once a Boolean two-dimensional matrix (same size as the initial grid) is returned with corresponding basin positions marked as true, the timer stops running. The duration of this classification is stored in a time variable.

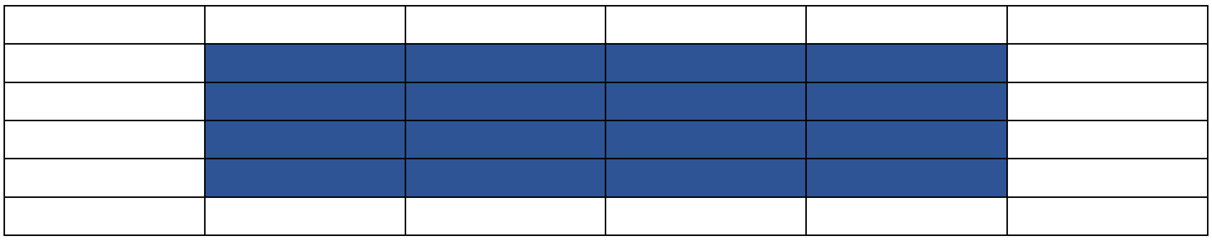


Figure 1. Given a 6x6 terrain, the processMatrix() method will only visit these cells

The printBasinsToFile() method simply writes the basins and the count to an output file specified by the user.

Parallel Program:

The parallel program is slightly different. Here I have two classes titled:

```
class ParallelBasins
```

```
class ParallelThreads extends RecursiveAction
```

The ParallelBasins class contains my main method and many of the same methods defined in the SequentialBasins method such as reading and writing to input and output files respectively. The main takeaway from this class is that it calls the invoke() method which starts the parallel part of the program.

As shown in the class definition above, the ParallelThreads class extends the RecursiveAction class which comes from the ForkJoinTask library. The ParallelThreads class has an overridden compute method which does not return anything (this is a property of the RecursiveAction class). The constructor takes in a two-dimensional version of the matrix we are processing as well as a one-dimensional one as well as the high and low values used to check length and split the work further. The 1-D array serves for splitting the array by rows. There is a check to see if the sequential cutoff is less than the portion of the matrix. If the portion (i.e high - low) is small enough to be processed, a processMatrix() method like the one in my SequentialBasins file will be called. If the sequential cutoff condition is not met, the program creates two new threads with a smaller portion to process as shown below:

```
ParallelThreads L= new ParallelThreads(grid,array,basins,low,(high + low) /2);

ParallelThreads R = new ParallelThreads(grid,array,basins,(high+low)/2, high);
```

To implement the divide and conquer algorithm we fork every child thread left child and recursively run compute on the right children. The left threads are then joined to aggregate the results into one Boolean grid where basins are categorized as 'true'. Each class of type RecursiveAction has its own variables and methods it uses to process its part of the grid.

Unique problem in sequential cutoff zone for parallel execution (Approach to solution):

While writing my code I faced quite a plethora of challenges that were non-existent in the sequential version of the program. First, since each thread would be processing different parts of the input grid, I needed each ParallelTheads class to have its own variables that would mark the following in each instance of the thread:

```
int startingRow
```

```
int startingCol
```

```
int finishingRow
```

```
int finishingCol
```

I had a big problem coordinating between where one thread would finish processing and where the next one would begin. Since I was splitting the array by row sometimes I would get duplicate results if two threads process the same line. This problem was solved using the following code block:

```
                    if(low == i * (grid[0].length) + 0)
                    {
                        startingRow = i;
                        //startingCol = j;
                    }
                    if(high - 1 == i * (grid[0].length) + j)
                    {
                        finishingRow = i;
                        finishingCol = j-1;
                    }
                }
            }
            if(low == 0)
            {
                startingRow = 1;
                startingCol = 1;
            }
            if(startingCol != 1)
            {
                startingCol = 1;
            }
            if(finishingRow == grid.length - 1  || finishingCol == grid[0].len
gth - 1)
            {
                finishingRow -= 1;
                finishingCol -= 1;
            }
```

This code block has several fixes/checks for various possible scenarios and sets the starting and ending locations. This is very important even if my code would have given me correct results because it avoids wasting resources by having two separate threads check the same location in the grid for basins. It is worth noting that my initial fix of switching from storing basins in an array of strings to an array of Booleans did fix my duplicates problem but didn't solve the problem of checking the same line twice (since it would just set the basin index to true twice). Correctly coordinating starting and ending points between threads was pivotal to producing faster, more efficient code. This mistake would have reflected in speed comparisons to my sequential program.

Experiment description:

To conduct the experiment, I first wrote a bash script to help automation of this process much easier. Script files can be found inside the experiment folder of the submitted zip file. This enabled me to run the SequentialBasins.java program 30 times and find an average for each of the run times. Moreover, I ran the Parallel version of the program using a similar script and found an aggregate of the time. The key difference with this portion of the experiment is the fact that I changed my sequential cutoff based on the number of datapoints. I also made several with a fixed sequential cutoff for each of the input sizes. I then varied the sequential cutoff across all sizes of input.

To check if my algorithm was producing correct basins, I compared my output files to expected results for each data size. Another way to check if my algorithm was performing appropriately was supplying it with a terrain with no basins and see what the result would be. Finally, I would say that the longer execution times as the data size increases is also a form of algorithm verification.

To measure the experimental speedup, I simply divided the time it takes to run serially with the time it takes to run parallelly as shown below. I did this for all the data sizes provided to get a sense of how speedup changes with data size/sequential cutoff.

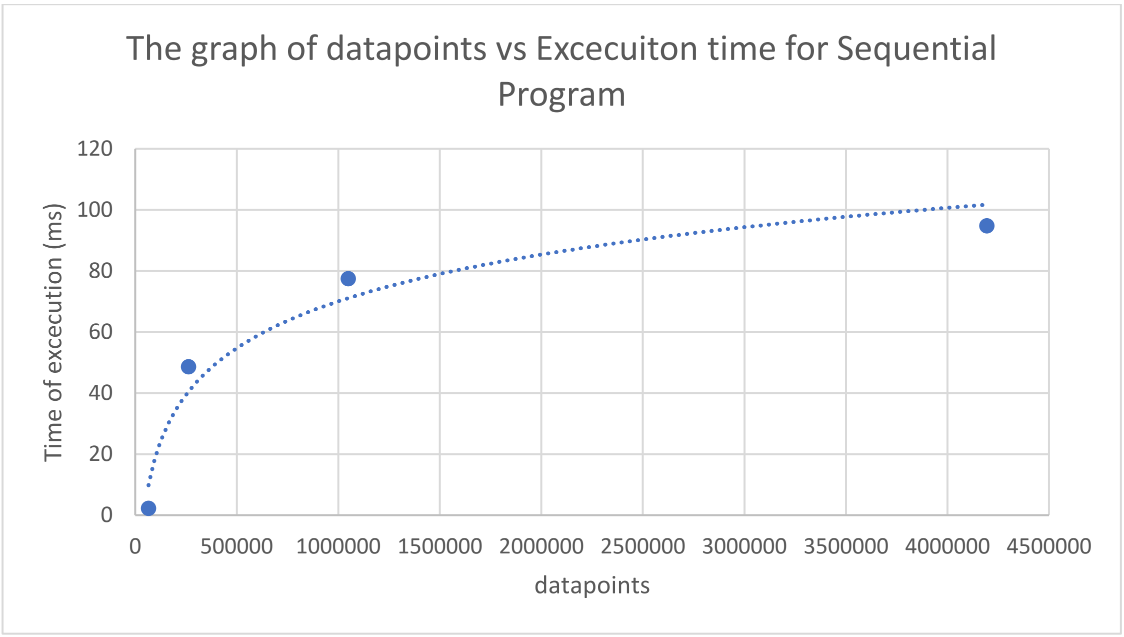
$$experimental\ speedup = \frac{serial\ runtime}{parallel\ runtime}$$

Results

The table below oulines my findings on a 2 core machine:

Datapoints	Average runtime(aggregated over 30 runs)
65536	2.229ms
262144	48.605ms
1048576	77.443ms
4194304	94.750ms

A graphical representation of the data above can be found below:

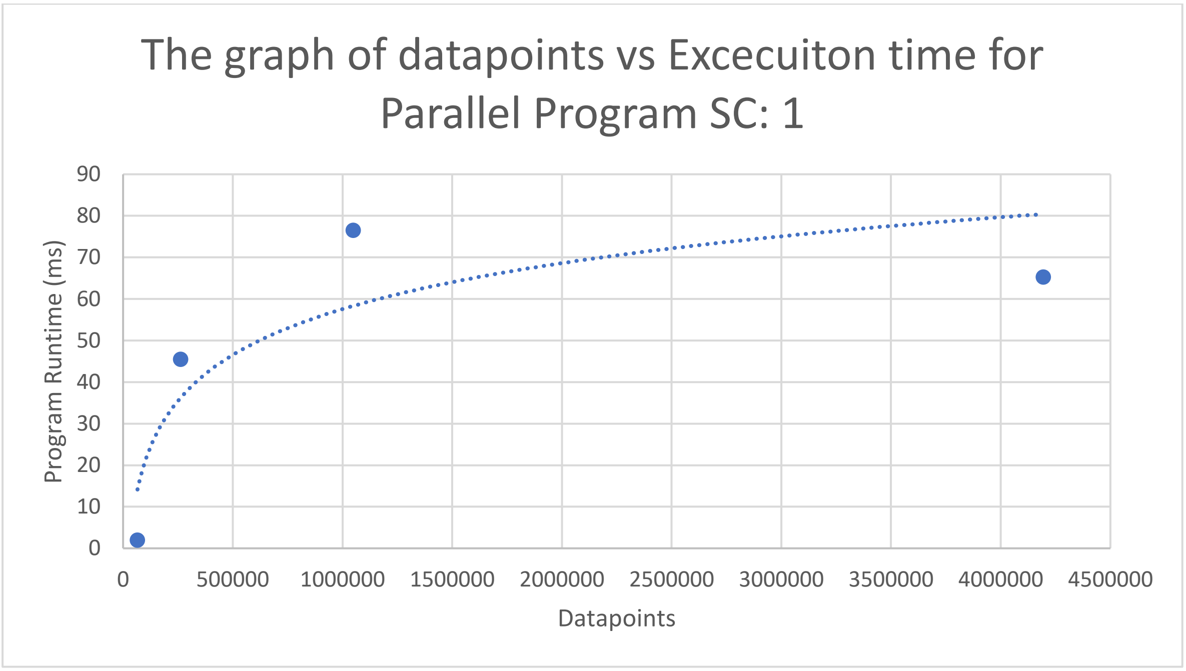


Parallel Threads Run:

For this portion of the test I set the sequential cutoff as a fraction of the total input data. I varied this fraction to find the ‘sweet spot’ which would yield the best performance throughout.

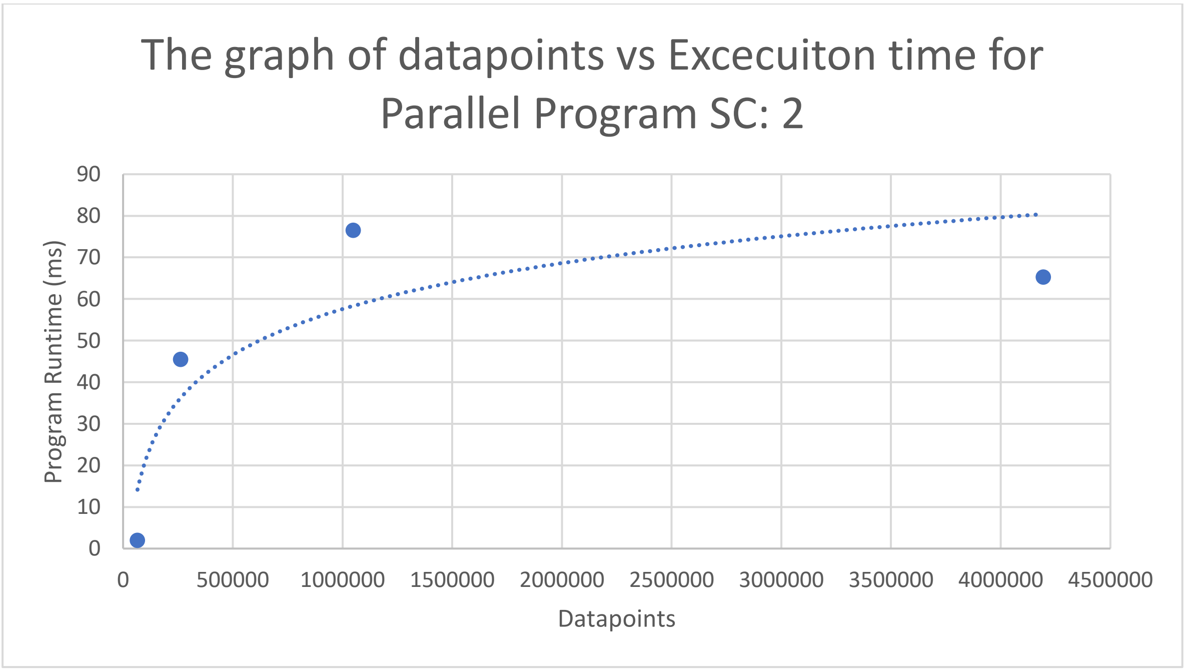
$$sequential\ cutoff = \frac{datapoints}{8}$$

Datapoints	Average runtime(aggregated over 30 runs)
65536	1.9458ms
262144	45.475ms
1048576	76.4745ms
4194304	65.241ms



$$sequential\ cutoff = \frac{datapoints}{4}$$

Datapoints	Average runtime(aggregated over 30 runs)
65536	1.7982ms
262144	40.475ms
1048576	69.420ms
4194304	52.241ms



For the calculation of the speedup I chose the to use the ‘optimal’ sequential cutoff which was:

$$sequential\ cutoff = \frac{datapoints}{4}$$

I used the values from this to calculate the speedup for each of the data sizes

Datapoints	Experimental Speedup
65536	1.2400
262144	1.2010
1048576	1.1200
4194304	1.8137

Discussion of Results

From the results we can see that there is not much of an improvement of results up to one million datapoints. We see a significant improvement in the speed up when we have about four million data points. From these results I would say that I think parallelization is worth it. In the day of the internet millions and billions of data is being generated every day. Imagine a product like Google, or Facebook. They probable receive many requests to their services daily and this small experiment shows that it is indeed worth perusing in those circumstances.

My maximum speed up is somewhat close to the theoretical speed up. I suspect that I will indeed get close to the theoretical speedup as my datapoints increase.

Conclusion

Here is a list of conclusions I have ascertained from conducting this experiment:

- Parallelism is better than a sequential approach as you increase data size
  - Sometimes this isn’t the case. It is simply faster to run your program sequentially. The real differences will be noticed as you increase you data size