DEONY BARRINGTON[BRRDEO001]

Assignment 1 [CSC2002S]

AUGUST 24, 2020

TABLE OF CONTENTS

Introduction..... 2
Methods..... 3
Results..... 5
Sequential Programming Graphs..... 5
Parallel Programming Graphs..... 7
Discusion..... 12
Conclusions..... 13
Extension to the assignment..... 14

INTRODUCTION

The aim of the project is to compare the time of execution using parallel algorithms on a range of data set sizes as well as a range of the number of threads. The idea is to see the effect on execution time as a result of the following things:

- Data set sizes
- Number of threads, i.e. the different sequential cut-off values.

These results will then be compared to the execution time of sequential algorithms on the same data sets, in order to see the speed differences in the execution times. When creating (and using) algorithms, they need to be as efficient and fast as possible. The conclusion of the assignment will allow us to see which algorithms (sequential or parallelism) is more efficient in analysing the data sets provided to us in this assignment.

In this assignment, data needs to be processed and minimum values needs to be found. That means that there are 2 important algorithms involved:

1. The divide-and-conquer algorithm to analyse the data parallelistic.
2. A linear search algorithm in order to analyse the data sequentially.

The hypothesis is that programming in parallel will significantly speedup the execution time to process the provided data.

METHODS

The purpose of the assignment is to see how parallel programming would increase the speed of execution. In order to see this, I decided to do the following:

I firstly created a program that run sequentially when receiving the data from a text file and stores it in a 2D array. The 2D array was then processed to find basins according to the conditions laid out in the assignment. The basins were then stored in an array list as they were found. I created 2 methods called tick() and tock() that times the execution of the application using the *System.currentTimeMillis()* method. The program was run 20 times for each datafile size (small, medium and large) that was provided. The sequential program was therefore run for a total of 60 times. For each of the datafile sizes, the execution times were recorded and measures of central tenancy for execution time was determined for each datafile size. Time taken to read in the text file was not included in the timing. A counter was also included to see how many basins was found for each datafile size, which would be used to aid in the determination of success or failure of the program.

After the sequential program ran successfully, I created the parallel program. The success of the sequential program (as well as the program that makes use of parallelism) was determined by the following criteria:

1. The program had to produce the correct number of basins found for each specific datafile size and had to correspond to the number of basins in the small_out.txt, med_out.txt and large_out.txt files provided.
2. After the correct number of basins had been determined, the program also had to find the same points as in the provided output files. This was checked manually.
3. The last criteria that had to be met in order for the program to be determined successful was that the order of the basins had to be in the same order as provided. This was also checked manually.

Once the sequential program was successful, I used the same insertion and search algorithms as used in the sequential program in order to find the basins in parallel.

The insertion algorithm was implemented in the construction method for the parallel program. The parallel program extended the RecursiveTask class with its return type being ArrayList<String> since I was returning an array list with the basins that has been found. The array was divided up into threads by its

rows using the divide and conquer method. At first the array is split into 2 rows, and if the number of rows was less than the sequential cut-off, then the rows between the lower boundary and upper boundary was processed sequentially. If it was more, the number of rows was split in half once more and another 2 threads was created. This continued recursively until each thread had an amount of rows to process that was less than the sequential cut-off. If it could be done sequentially, the findBasins method was called. This method finds the basins in the rows between the lower bound and the upper bound. The execution time for the compute() method to run was recorded as execution time.. The program was run for a total of 900 times as follows:

1. Sequential cut-off value: 100
   Small_in: 100 times
   Medium_in: 100 times
   Large_in: 100 times

2. Sequential cut-off value: 200
   Small_in: 100 times
   Medium_in: 100 times
   Large_in: 100 times

3. Sequential cut-off value: 300
   Small_in: 100 times
   Medium_in: 100 times
   Large_in: 100 times

I tested the sequential program on laptop with specs as shown in figure 1(a) and the parallel program on both laptops with specs in figure 1(a) and figure 1(b)

Base speed:	1.99 GHz
Sockets:	1
Cores:	2
Logical processors:	4
Virtualization:	Enabled
L1 cache:	128 KB
L2 cache:	512 KB
L3 cache:	3.0 MB

Figure 1(a): Lenovo Laptop Specs

Base speed:	2.81 GHz
Sockets:	1
Cores:	4
Logical processors:	8
Virtualization:	Enabled
L1 cache:	256 KB
L2 cache:	1.0 MB
L3 cache:	6.0 MB

Figure 1(b): Dell Laptop Specs

RESULTS

The results for the 2 different types of programming methods are as follows:

SEQUENTIAL PROGRAMMING

Visual Representations Of Execution Times Of Different Dataset Sizes

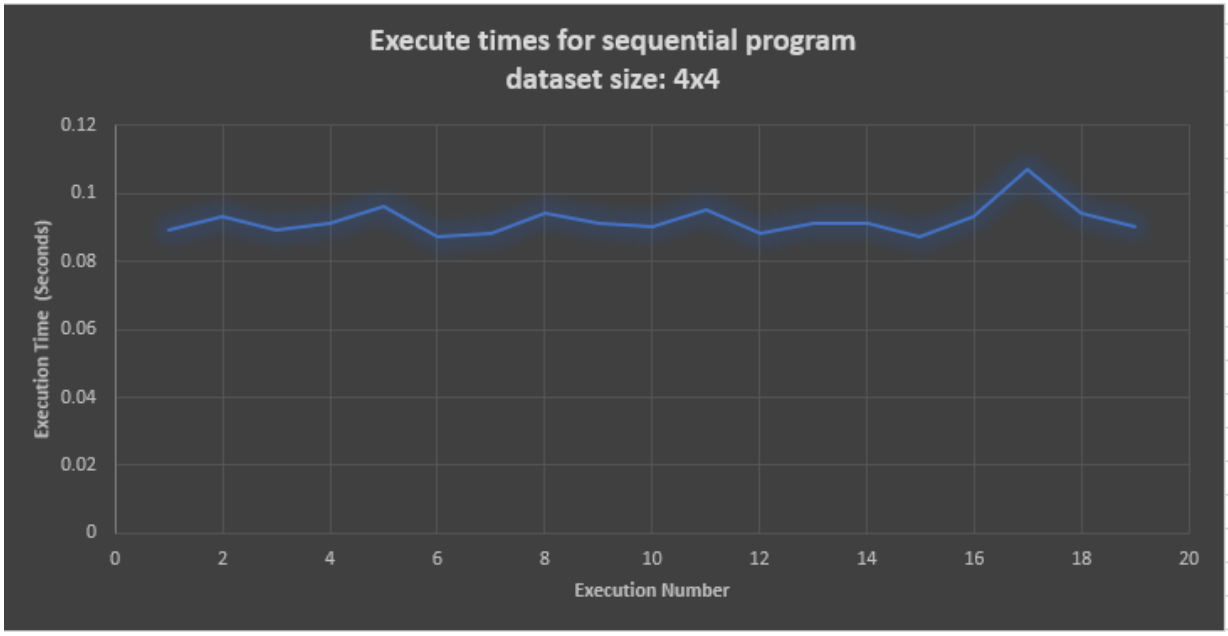

Figure 2(a): Graph showing execution time for 4x4 datafile size

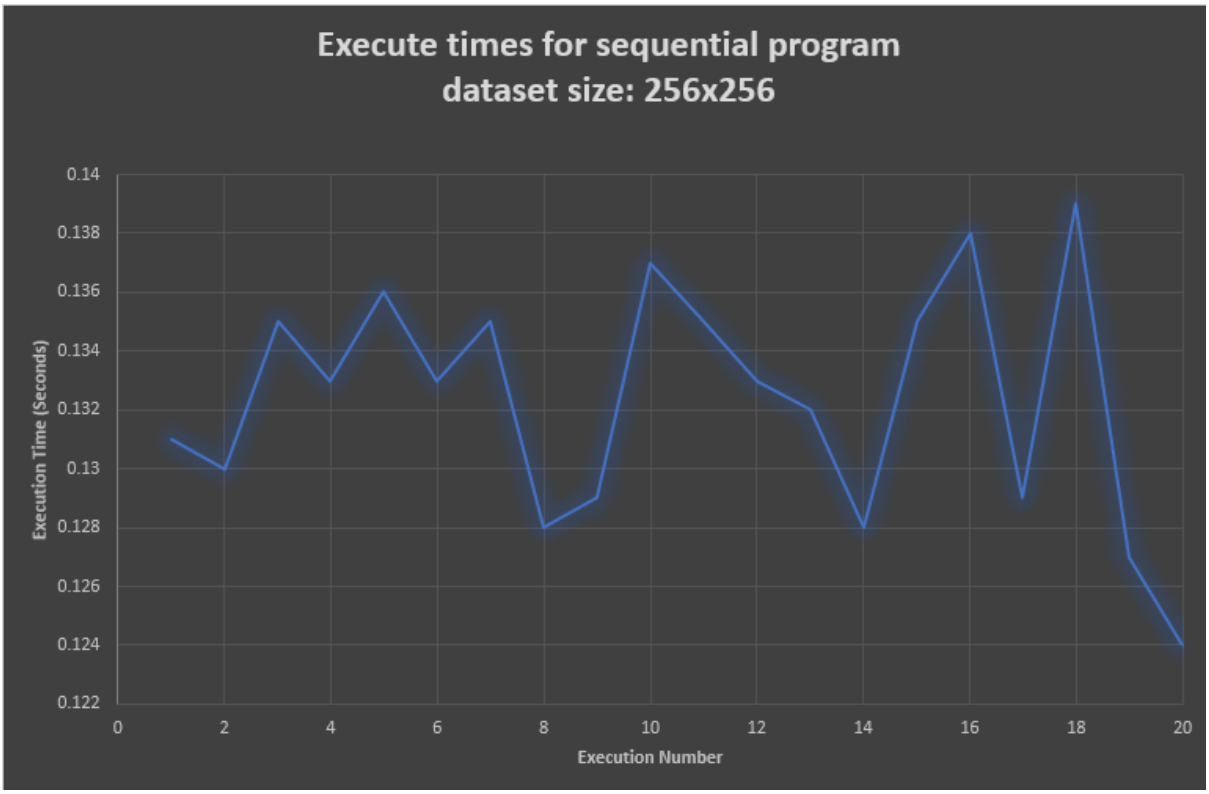

Figure 2(b): Graph showing execution time for 256x256 datafile size

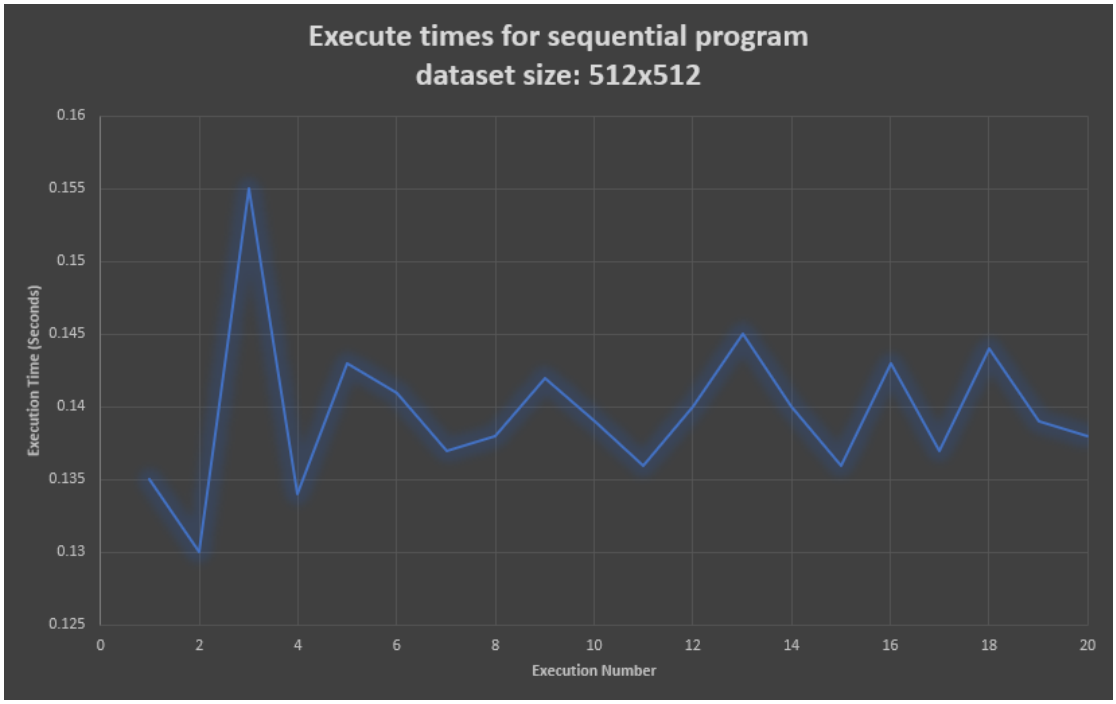

Figure 2(d): Graph showing execution time for 1024x1024 datafile size

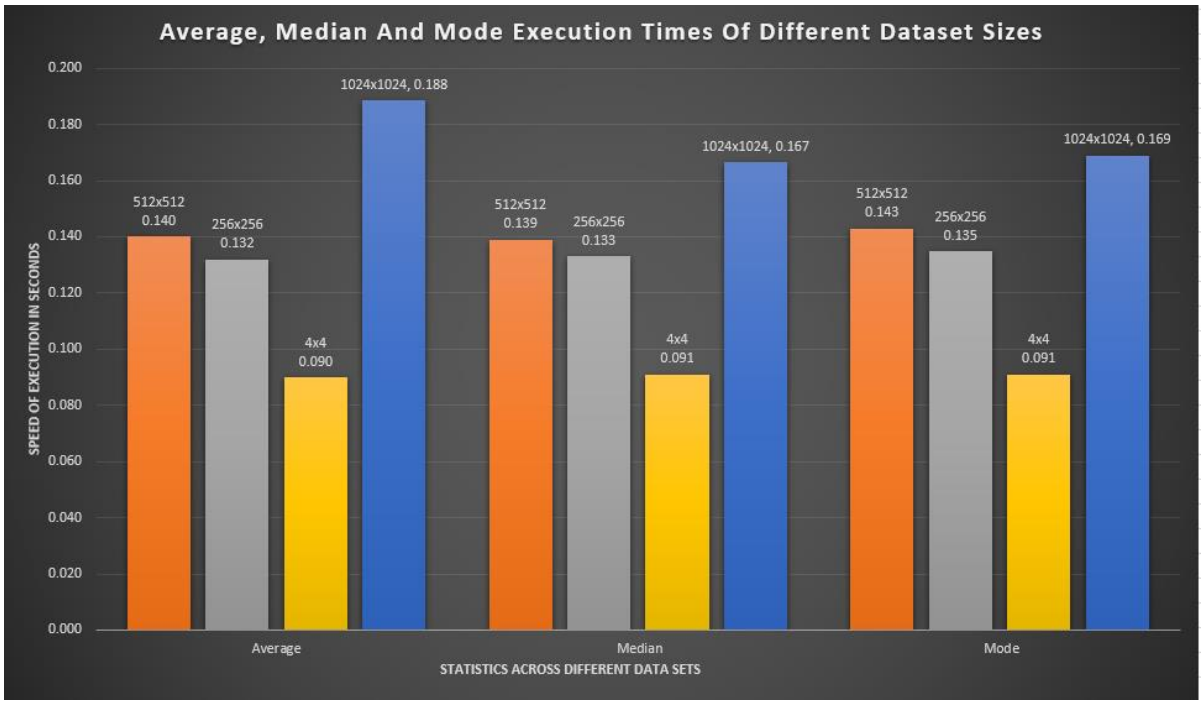

Figure 2(e): Graph showing measures of central tenancy for different datafile sizes

Graphs showing execution times for different datafile sizes with sequential cut off value being 100

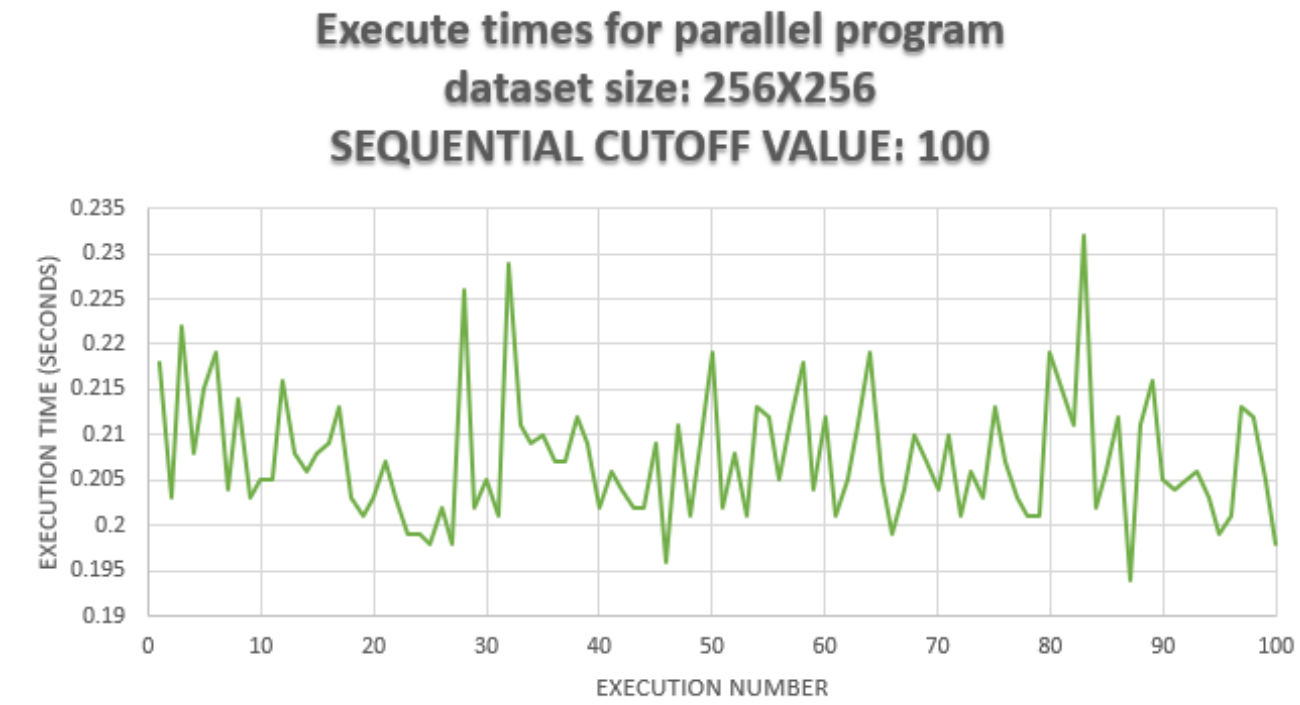

Figure 3(a): Graph showing execution time for 256x256 datafile where sequential cut-off value is 100

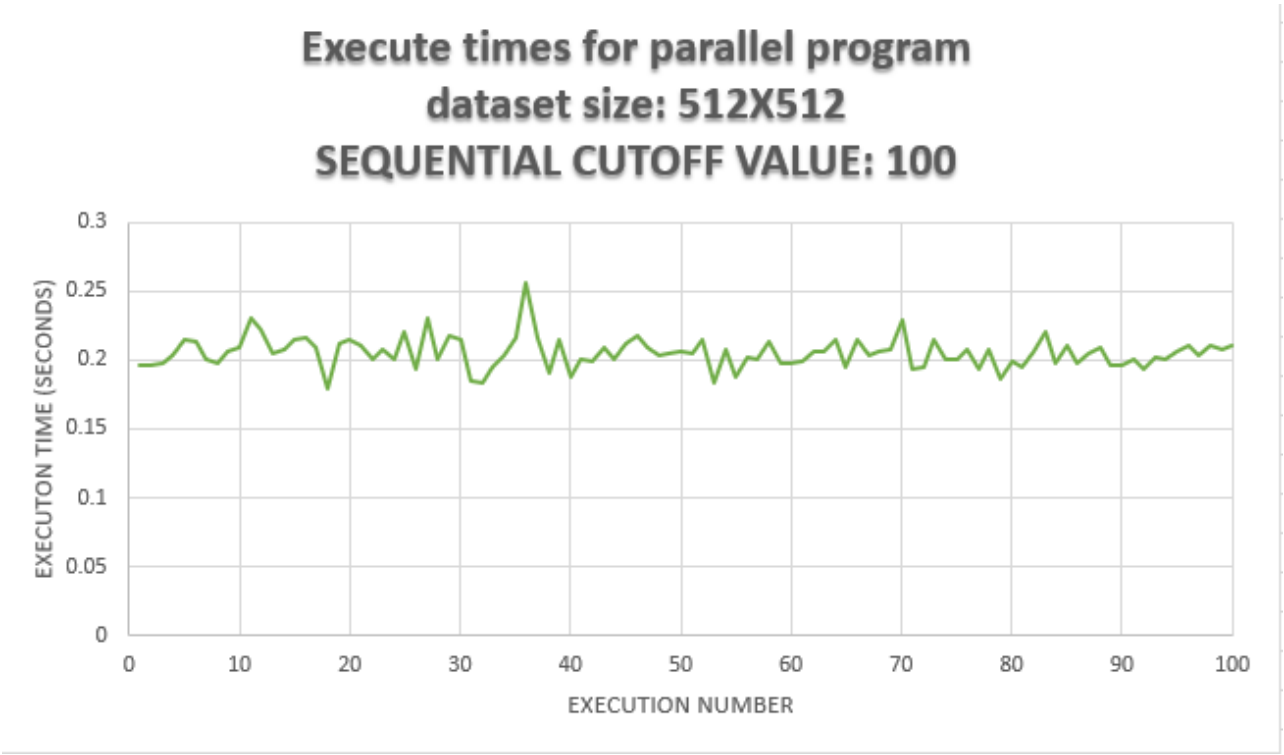

Figure 3(b): Graph showing execution time for 512x datafile where sequential cut-off value is 100

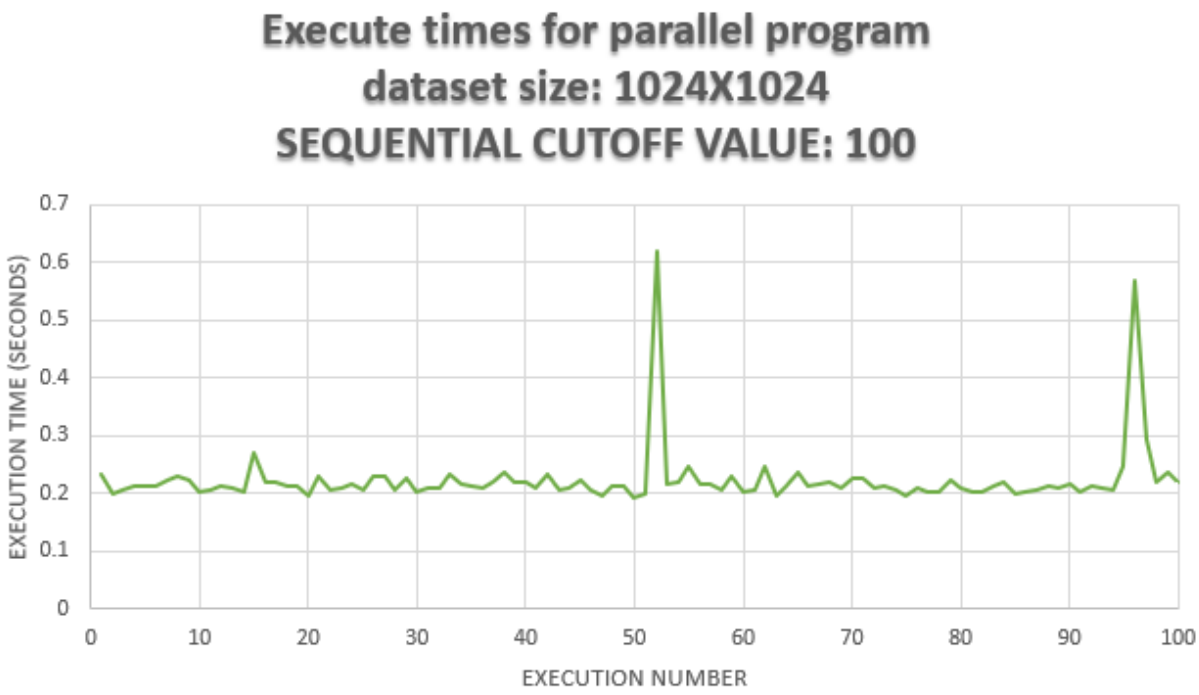

Figure 3(c): Graph showing execution time for 1024x1024 datafile where sequential cut-off value is 100

<u>Graphs showing execution times for different datafile sizes with sequential cut off value being 200</u>

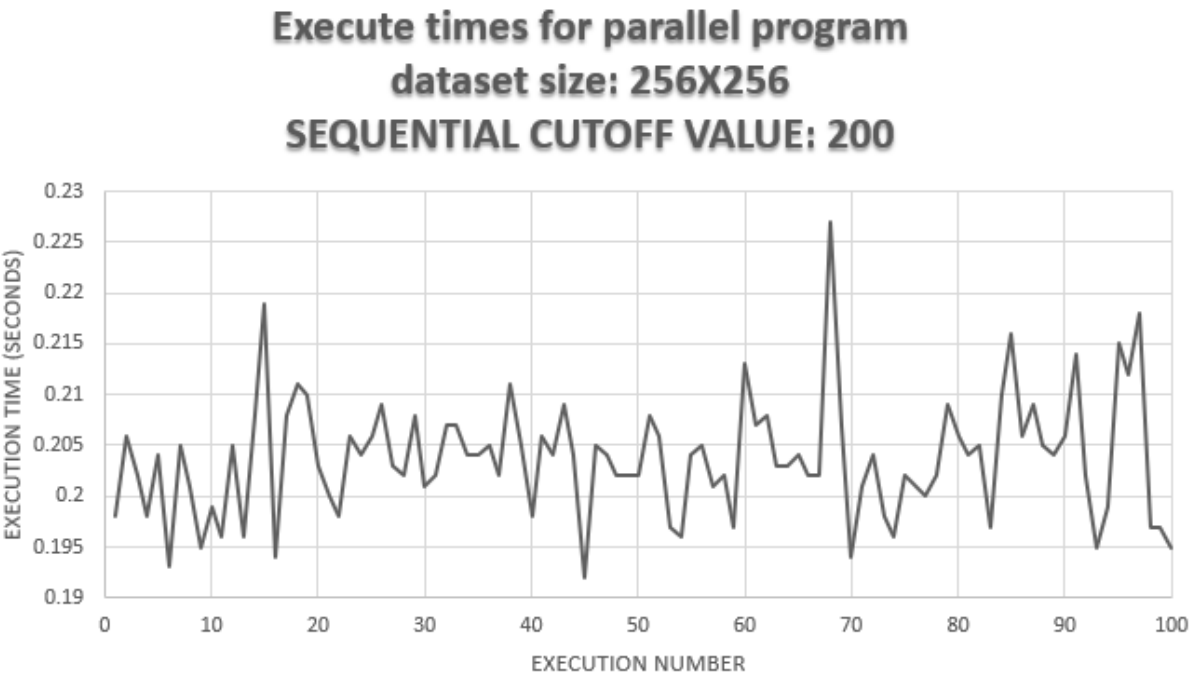

Figure 4(a): Graph showing execution time for 256x256 datafile where sequential cut-off value is 200

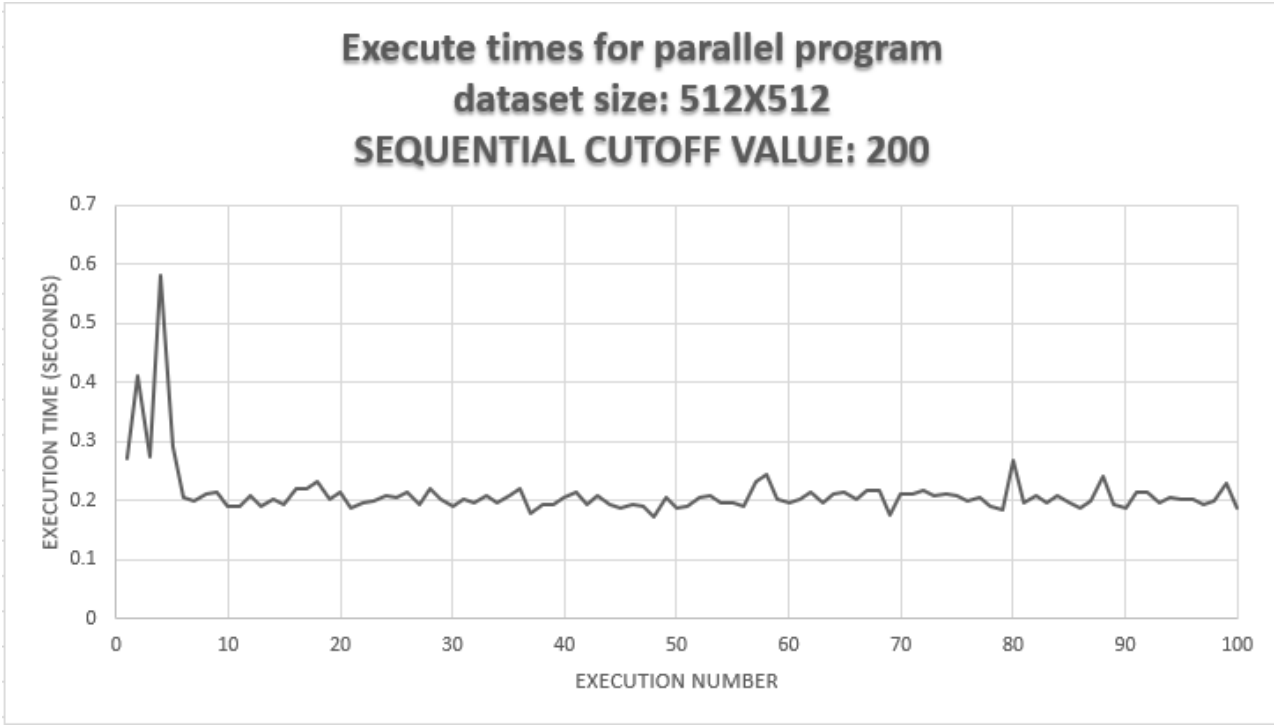

Figure 4(b): Graph showing execution time for 512x512 datafile where sequential cut-off value is 2

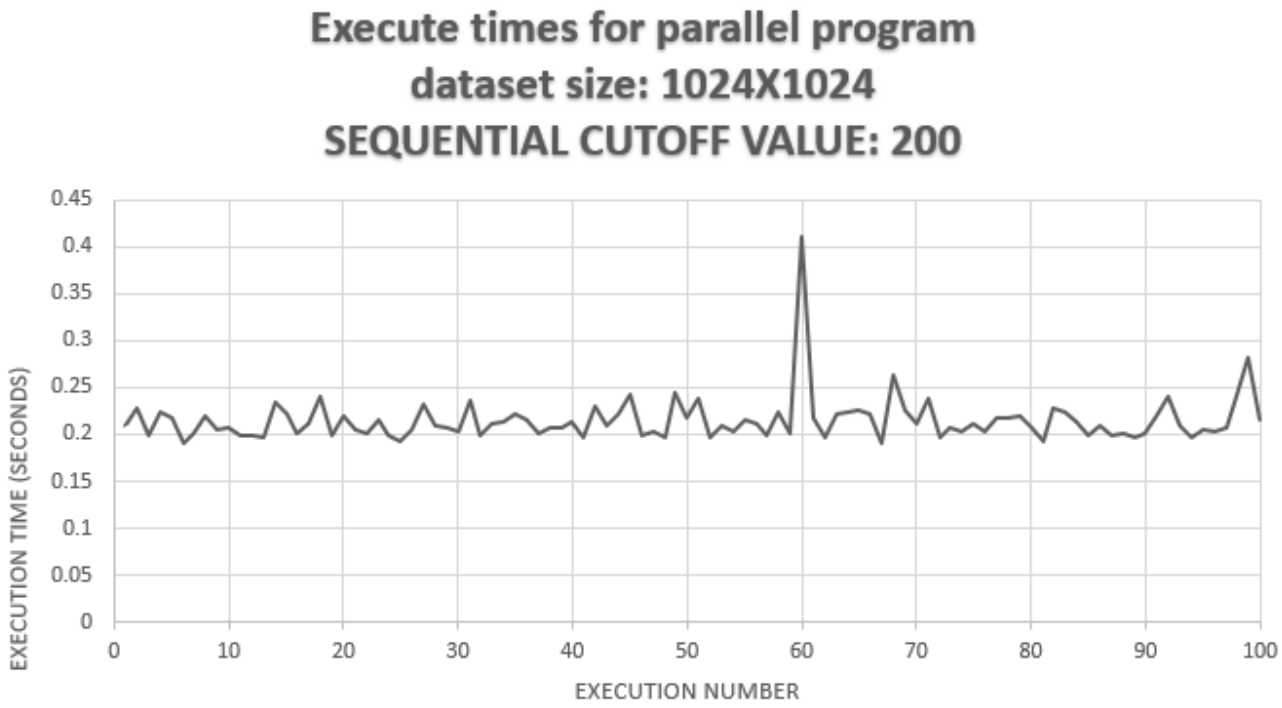

Figure 4(c): Graph showing execution time for 1024x1024 datafile where sequential cut-off value is 200

Graphs showing execution times for different datafile sizes with sequential cut off value being 300

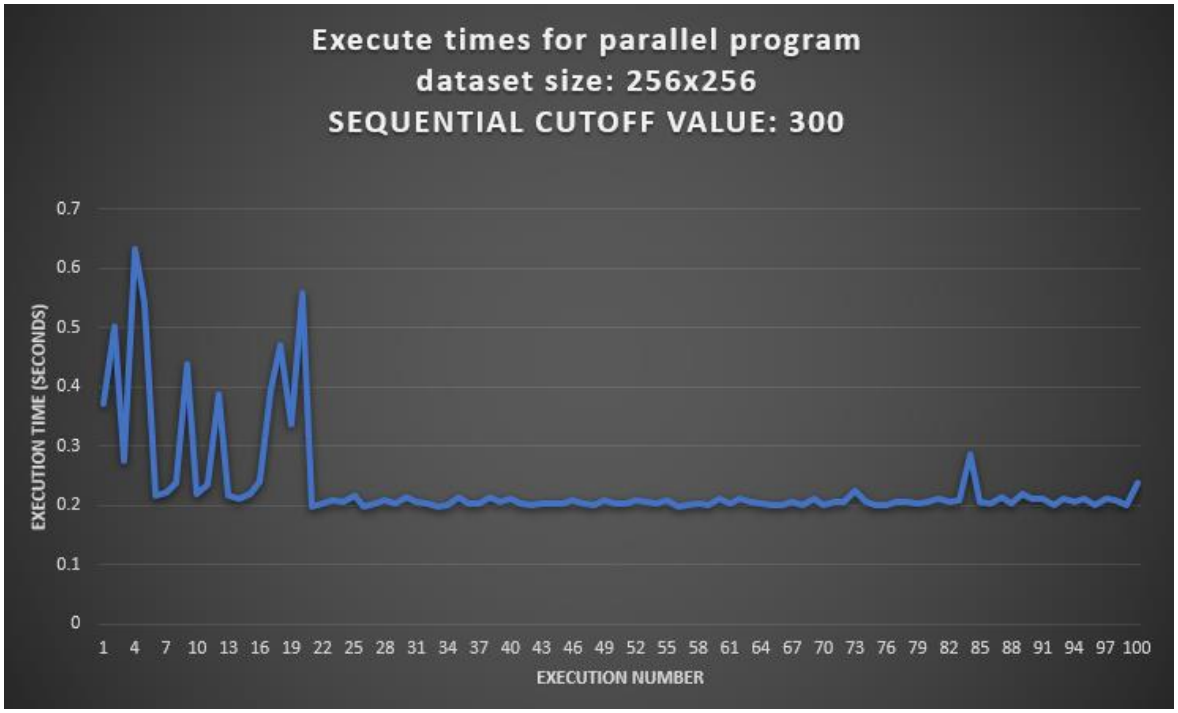

Figure 5(a): Graph showing execution time for 256x256 datafile where sequential cut-off value is 300

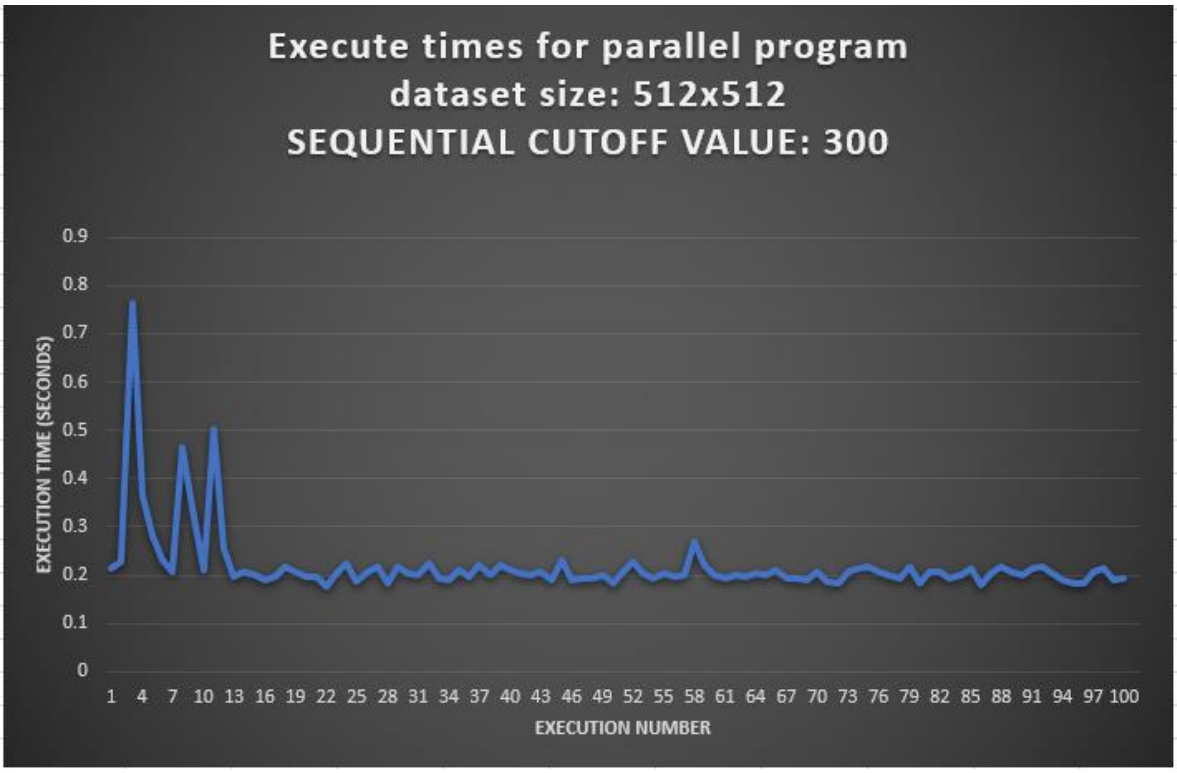

Figure 5(b): Graph showing execution time for 256x256 datafile where sequential cut-off value is 300

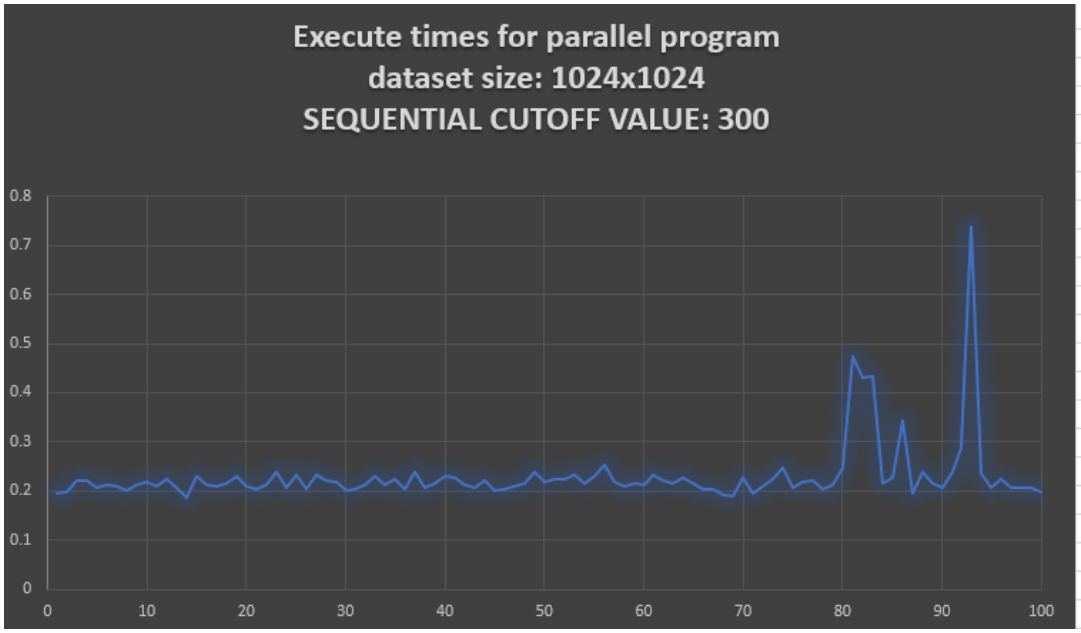

Figure 5(c): Graph showing execution time for 1024x1024 datafile where sequential cut-off value is 300

SUMMARY OF EXECUTION TIMES

SERIAL

256X256	*512X512*	*1024X1024*	*LAPTOP USED*
0.132	*0.140*	*0.188*	*LENOVO*
0.036	*0.042*	*0.043*	*DELL*

PARALLEL(LENOVO LAPTOP)

SCO VALUE	*256X256*	*512X512*	*1024X1024*
100	*0.207*	*0.205*	*0.223*
200	*0.204*	*0.212*	*0.215*
300	*0.236*	*0.219*	*0.230*

PARALLEL(DELL LAPTOP)

SCO VALUE	*256X256*	*512X512*	*1024X1024*
100	*0.025*	*0.053*	*0.073*
200	*0.024*	*0.047*	*0.065*
300	*0.0.21*	*0.039*	*0.062*

DISCUSSION

Both program were first run in Ubuntu (virtual box) which was assigned only 1 core upon set up. This resulted in a slower performance by the parallel program because it behaved in the same way as the serial program- only 1 (or 2) threads could be executed at a time and in addition to this delay in threads being executed, the parallel program has to compare the lower and upper boundaries to the sequential cut-off value which further delayed execution time. The sequential program took between 0.132 and 0.188 seconds on average to execute, depending on the datafile size, where the parallel program took on average between 0.204 and 0.236 seconds to execute, varying according to the sequential cut-off value as well as the datafile size. Once I realised that the true efficiency potential could not be seen on this laptop, I ran both programs on another laptop with more cores and better processing power. The sequential program ran more than 3 times faster and the parallel program ran 5 – 10 times faster. The difference in execution time between the parallel and serial program was not much, but it was there.

Whether using parallelization is worth it to tackle this particular problem in java, depends fully on the capabilities of the machine on which it is running. This is evident in the performance results that I obtained while experimenting with the 2 programs on the 1 different machines. On the one machine it was a complete waste of time and on the other performance increase was minimal, however it was still there. It therefore completely depends on the machine architecture.

The parallel program performed at its best with a sequential cut-off value of 300 and the small data text file. It had the highest increase ratio between all the tested sequential cut-off values. The program was also tested with a sequential cut off value of 400 and 500, however, the program performance was slower.

CONCLUSIONS

While doing the assignment and running the tests, I came to a few conclusions regarding using parallelization

- Parallelization would prove most useful when processing data on extremely powerful machines. Data such as astronomical, satellite and weather data that are analysed on extremely powerful machines would benefit from parallelization more than anything or anyone else. This is not limited to the mentioned examples, but includes any data that is analysed on extremely powerful machines.
- Making use of parallelization when creating programs for people whose machine power is unknown, is risky. Parallelization can cause slower performance, evident in the results I have obtained, which defeats it's purpose. Sequential programs seems like a better option for unknown architecture.
- Multithreading can only be used to its full potential if a person fully knows what they are doing. Perhaps my programs weren't as efficient because I didn't know how to use the Fork/Join framework to reach its fullest potential.

The conclusions that I have made are valid based on the data that I have accumulated, however can be disproved at any point. Although I've come to these conclusions myself, I cannot fully say that it is true, because I haven't learnt all ways in which to optimize parallelization. These are simply conclusions that I have made based on the data that I have accumulated and my own experience up to this point in time.

EXTENSION TO THE ASSIGNMENT

After seeing a negligible increase in performance, I tried running the programs on one more machine with the following processing power:

The results were as follows:

execution time: 0.016 ms	execution time: 0.015 ms
execution time: 0.016 ms	execution time: 0.032 ms
execution time: 0.031 ms	execution time: 0.015 ms
execution time: 0.016 ms	execution time: 0.032 ms
execution time: 0.031 ms	execution time: 0.031 ms
execution time: 0.016 ms	execution time: 0.016 ms
execution time: 0.031 ms	execution time: 0.015 ms
execution time: 0.031 ms	execution time: 0.015 ms
execution time: 0.015 ms	execution time: 0.031 ms
execution time: 0.031 ms	execution time: 0.016 ms
execution time: 0.016 ms	execution time: 0.015 ms
execution time: 0.016 ms	execution time: 0.032 ms
execution time: 0.031 ms	execution time: 0.015 ms

The difference in execution time was even more insignificant.

I decided to create the parallel program by extending the Thread class. The sequential program ran at an average of 22ms and ran about 5 times faster in parallel, with an average of 5ms. Due to time constraints, the parallel program extending the thread class was not tested on the Lenovo or Dell laptop. The reason for this significant increase is unknown to me, but it definitely made a difference. The execution times were as follow:

```
Sequential: large-in.txt
Done processing, execution time: 22.6598 ms
Done processing, execution time: 22.8276 ms
Done processing, execution time: 22.8414 ms
Done processing, execution time: 22.4701 ms
Done processing, execution time: 28.3399 ms
Done processing, execution time: 22.3629 ms
Done processing, execution time: 22.6719 ms
Done processing, execution time: 22.8205 ms
Done processing, execution time: 22.5787 ms
Done processing, execution time: 22.8676 ms
Parallel:5 Threads, large_in.txt
Done processing, execution time: 1.0565 ms
Done processing, execution time: 3.4666 ms
Done processing, execution time: 4.2097 ms
Done processing, execution time: 6.676 ms
Done processing, execution time: 3.4946 ms
Done processing, execution time: 3.8285 ms
Done processing, execution time: 1.9438 ms
Done processing, execution time: 6.9957 ms
Done processing, execution time: 4.3258 ms
```

Git-Log:

commit e666298048573afdd09c710b5b4ba9542b5dc1dd (HEAD -> master)
Author: BRRDEO001 <deonyb23@gmail.com>
Date: Sun Aug 23 15:20:02 2020 +0200

Added Makefile

commit befe9ca5de34a4ae0f6c62a515898a6b1c5bfc54
Author: BRRDEO001 <deonyb23@gmail.com>
Date: Sun Aug 23 15:19:30 2020 +0200

Added all .java files