Parallel Programming with the Java Fork/Join framework: Terrain Classification

MGWGIF001

23 August 2020

Contents

1 Introduction

1.1 Background

In this modern day society, the scanning of the earth's surface for various purposes has become common place. For the purposes of our investigation, attention was placed on the bare earth survey which map the height of the earths surfaces stripped of buildings and trees. When performing these scans, the datasets generated can be quite large with complex analysis associated with them as such, they make good candidates for parallel computing.

With the above in mind, an example analysis is determining how water flows across the landscape in order to protect against flooding and manage crops. Before this can be performed however, the input terrain must be scanned to find local minima i.e. small scale basins where water might accumulate

1.2 Aim

For our assignment, we are to investigate the notion of paralyzing the implementation of determining the local minima. For this investigation, we seek to answer the following questions

- Is it worth using parallelization (multithreading) to tackle this problem in Java?
- For what range of data set sizes will the parallel program perform well?
- What is the maximum speedup obtainable with the parallel approach? How close is this speedup to the ideal expected?
- What is an optimal sequential cut-off for this problem?
- What is the optimal number of threads on my system architecture?

1.3 Explanation of Parallel Algorithm Used

1.3.1 Example of determining basin

To explain the parallel algorithm used, let us consider the determining of a basin for a trivial example terrain show in Figure 1 which contains a grid of land sections of varying heights in meters.

1.00	0.90	0.95	0.80
1.00	0.95	0.90	0.80
0.85	0.60	0.80	0.75
1.00	1.00	1.00	1.00

Figure 1: Trivial Input Terrain Example

If we are to observe the grid, we can see that we have a local basin at $(2, 1)$. The reason for this, is that if we increase the value 0.60 by some offset 0.01 to allow for some noise in the data, we observe that it is smaller than all its neighbours which i.e. 1.00, 0.95, 0.90, 0.80, 1.00, 1.00, 1.00, 0.85. As an aside, of particular not is that, all the cells in the boundary of the grid are not considered to make things slightly easier.

1.3.2 Parallel Algorithm Used

From the above example, we can see that as the grid gets larger and larger, the determining of the basin is effectively on a point by point basis as only the immediate local neighbours are checked to determine whether it is a basin or not. As such, this problem is a good candidate for the implementation of the Map Algorithm.

For this algorithm to be implemented correctly and not be naively parallel, we see that we need to use a divide and conquer approach were we divide the grid into 4 equal section for each iteration as shown in Figure 2.

1.00	0.90	0.95	0.80
1.00	0.95	0.90	0.80
0.85	0.60	0.80	0.75
1.00	1.00	1.00	1.00

Figure 2: Applying Divide and Conquer to Trivial Example

Upon splitting the work, we see we have 4 smaller sub-threads are subsequently divided until the sequential limit is reached and the sub-grids are processed. In this trivial example, we only need to divide once and each thread can then check one grid location which is (1,1) for the top left, (1,2) for the top right, (2,1) for bottom left and finally (2,2) for bottom right. It is important to note that each thread still has access to the whole grid as it needs to check all neighbours but, it is restricted to checking only the points within its own sub-grid.

1.4 Expected Speedup/Performance

From the above discussion, we can see that algorithm being implemented still has the same end to end tree structure with the only difference being that each node in the Directed Acyclic Graph(DAG) has 4 children as visualized in Figure 3.

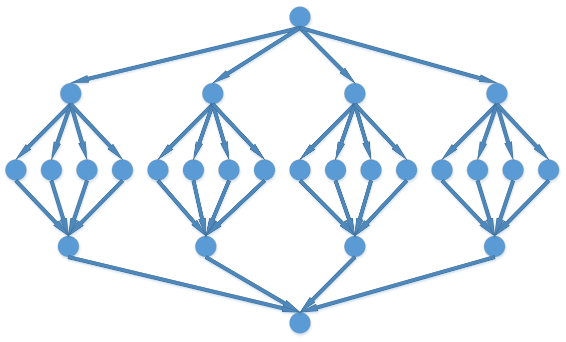

Figure 3: DAG for Parallel Algorithm

From the above, if we can see that for if the we have n processes, then

$$Work, T_1 = O(n)$$

For determining the span, it is slightly trickier, however, we know that for an n-array tree, with q leaf nodes, the height, h, is given by

$$h = \log_n(q)$$

therefore,

$$Span, T_\infty = 2\log_n(q) = O(\log(q))$$

As such, we not that

$$Speed\ up = \frac{T_1}{T_\infty} = \frac{n}{\log(n)}$$

From this, we can conclude that theoretically, this problem is worth making a parallel implementation of.

2 Methods

2.1 Approach

2.1.1 Determining Sequential Solution

For the sequential solution, a runner class by the name 'SerialRunner.java' was created where the main method is located. The entirety of the code is located there. The basic idea was to use 2 nested for-loops starting at position $(1,1)$ with the indices at position $(N_{row} - 2, N_{col} - 2)$ where N_{row} and N_{col} is the number of rows and columns respectively.

For the checking, the code moves to each point, adds the offset and check if the resulting value is smaller than its 8 neighbours. If it is, the index location is stored in a temporary ArrayList and it then moves on to the next location. This is repeated until the entire grid is finished the final output stored in the ArrayList is outputted.

2.1.2 Determining Parallel Solution

As mentioned above, the basic idea revolved around a task being sub-divided into four smaller subtasks. Since I was expecting to work with a large number of threads, the Java Fork/Join Framework was used. From here, the tasks would keep subdividing into smaller subtasks until the dimensions of the sub-grid are smaller than or equal to the sequential limit. Of note here, the code assumes a square grid hence the sequential limit for the row and columns of the sub-grid is the same.

Upon reaching the sequential limit, the same approach used for the sequential solution mentioned above was implemented. The only difference being that the ArrayLists were initially created for each individual thread during the fork section of the algorithm and then they get combined to form one final ArrayList at the join segment resulting in only one final list containing all the basins located by the individual threads. For this implementation, two classes were created which are ‘ParallelBasinClassify.java’ and ‘ParallelBasinRunner.java’. Of the two classes created, ‘ParallelBasinRunner.java’ contains the main method that contains the ForkJoinPool whilst ‘ParallelBasinClassify.java’ is the class that extends the RecursiveTask class.

2.2 Validation

For the validation of the program, the sample input output files provided with the assignment were used as the checkers. A since the parallel solution does not guarantee the order in which the output will come out in, a method called ‘checkMatchingResults’ located in ‘HelperMethods.java’ was written. This method initially checked if the found basins number is the same as the expected and then checks if all the basins found are the ones that are to be expected. Upon writing this method, it was then added as a validation check when running the program in the runner class.

2.3 Timing Algorithm

For the determining of the run times, the ‘System.currentTimeMillis’ method was heavily used. For this, two functions named ‘tick’ and ‘tock’ were written which are located in ‘HelperMethods.java’ which get the start time and calculate the runtime respectively. Upon creating these functions, they we placed just before the comparison module.

Of note is that, to allow for a more reliable time, each comparison operation for each sequential cut-off was repeated 3 times and the average time was used.

2.4 Measuring Speedup

For measuring the speedup, 3 pieces of information were extracted from the code

- The runtime for a certain sequential cut-off (ranging from 1 to 100)
- The number of threads created for a certain sequential cut-off (ranging from 1 to 100)
- The runtimes for sequential solution

For the experiment, 3 datasets categorised as shown in Table 1 were tested.

Table 1: Testing Dataset Categories

Dataset Category	Dataset size
Small	256 * 256
Medium	512 * 512
Large	1024 * 1024

2.5 Machine Architecture

For the testing purposes, the following single multi-core machine architecture was used.

Table 2: Testing Machine Architecture

Processor	Intel® Core™ i5-8250U CPU @ 1.60GHz (8 CPUs), ~ 1.8GHz
Memory (RAM)	11.9 GB
Operating System	Windows 10 Home Single Language 64-bit

2.6 Interesting Problems/Difficulties Encountered

An interesting problem that occurred to me was when I was trying to time my function. Initially, I implemented the timing algorithm using Float datatypes however, because my algorithms were so fast, I ended up with many cases were the established time was so small that it got rounded off to 0. As such, to help increase the accuracy, I ended up using doubles.

3 Results and Discussion

3.1 Runtimes for different datasets

For the experiment, the runtimes of the 3 datasets were measured and Figure 4 show the results.

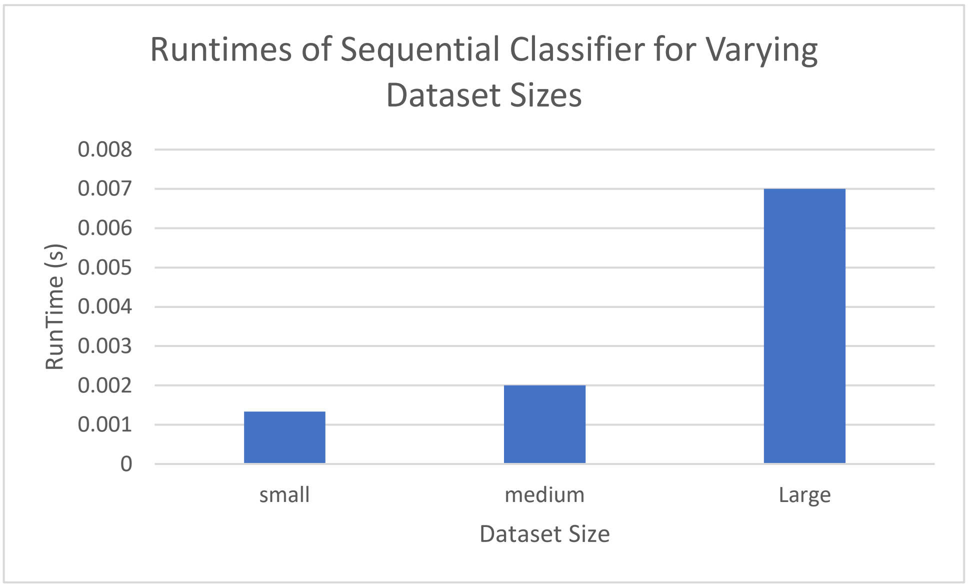

Figure 4: Runtimes for sequential solution

From the above, it can be seen that as the dataset increases, so too does the runtime as is to be expected.

Upon getting the runtimes for the sequential solution, the runtimes for the parallel solution were determined for different sequential cut-off limits ranging from 1 to 100 as shown in Figure 5

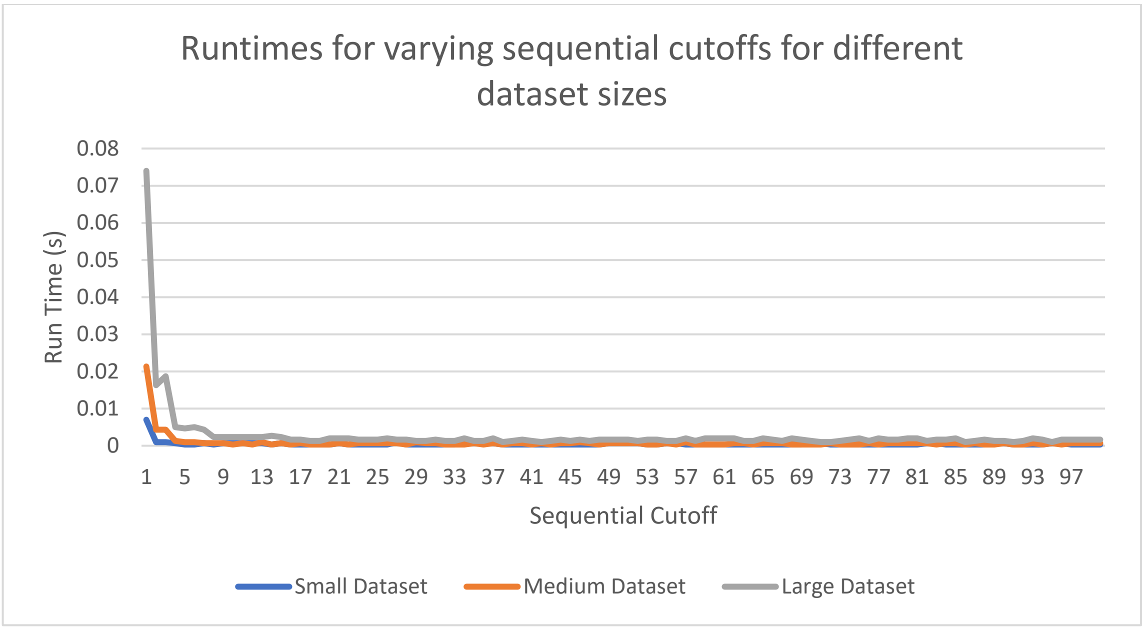

Figure 5: Runtimes for parallel solution

From the above plots, it was noted how quickly the runtime reduces and how in general, the sequential cut-off of 1 was the least optimal.

3.2 Speedup Graphs

3.2.1 Speedup Graphs with respect to Sequential Cut-off

Having obtained the runtimes, it was necessary to assess how much of a speed up was obtained as such Figure 6 to Figure 7 shows the speedups with respect to the sequential cut-off for varying dataset sizes.

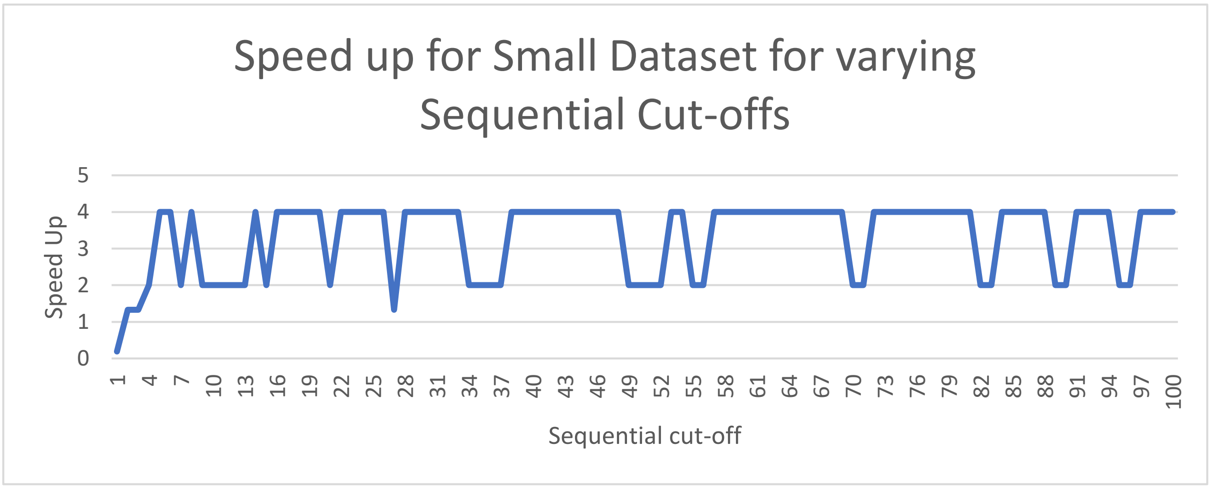

Figure 6: Speed Up for small dataset for varying sequential cut-offs

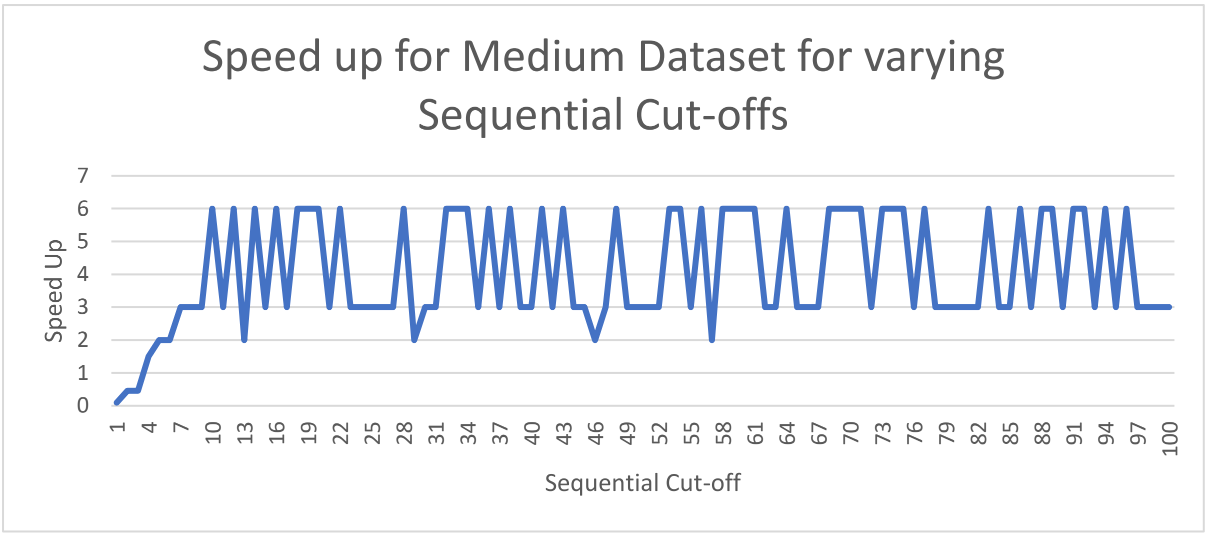

Figure 8: Speed Up for Medium Dataset for varying sequential cut-offs

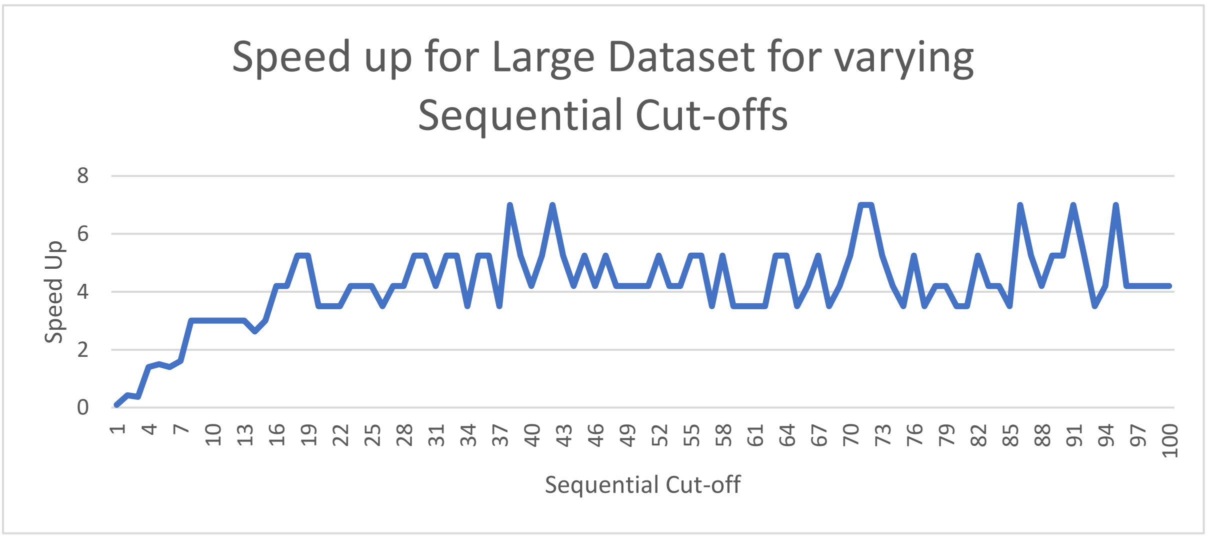

Figure 7: Speed Up for Large Dataset for varying sequential cut-offs

From the above plots, the below table was obtained which summaries the observations seen

Table 3: Summary of optimal Sequential Cut-offs

Dataset Size	Max Speed Up	Optimal Sequential Cut-off
Small	4	5 – 8
Medium	6	10 – 12
Large	5	19 – 30

3.2.2 Speedup Graphs with respect to Number of threads

Upon Assessing the speed up with respect to the sequential cut-offs, it was also checked with respect to the number of threads generated. From this, Figure 9 to Figure 11 were obtained.

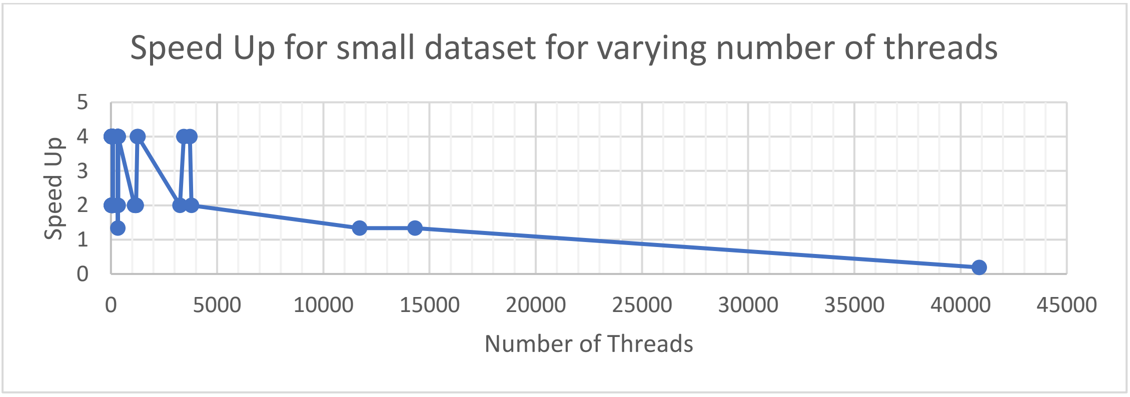

Figure 9: Speed Up for small dataset for varying number of threads

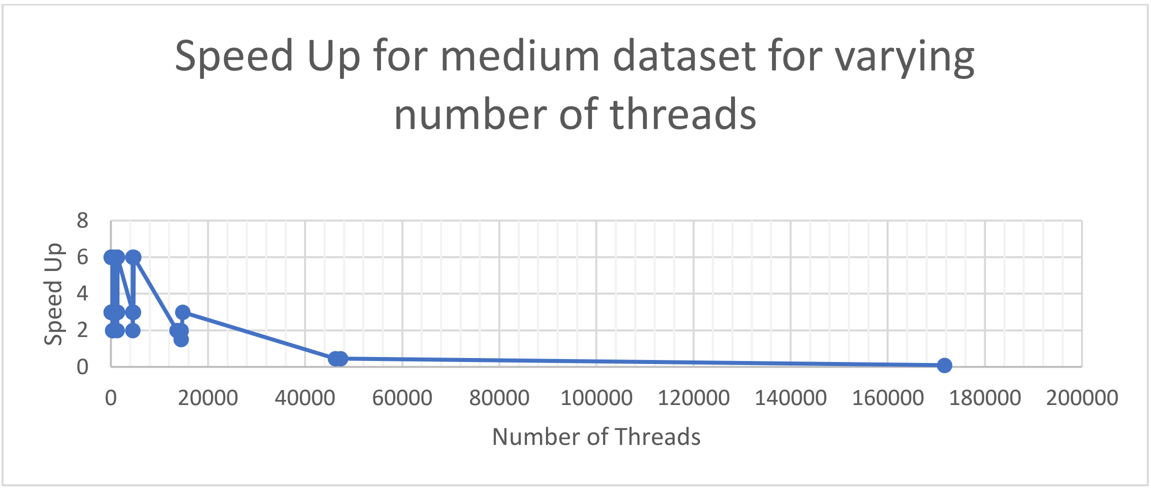

Figure 10: Speed Up for medium dataset for varying number of threads

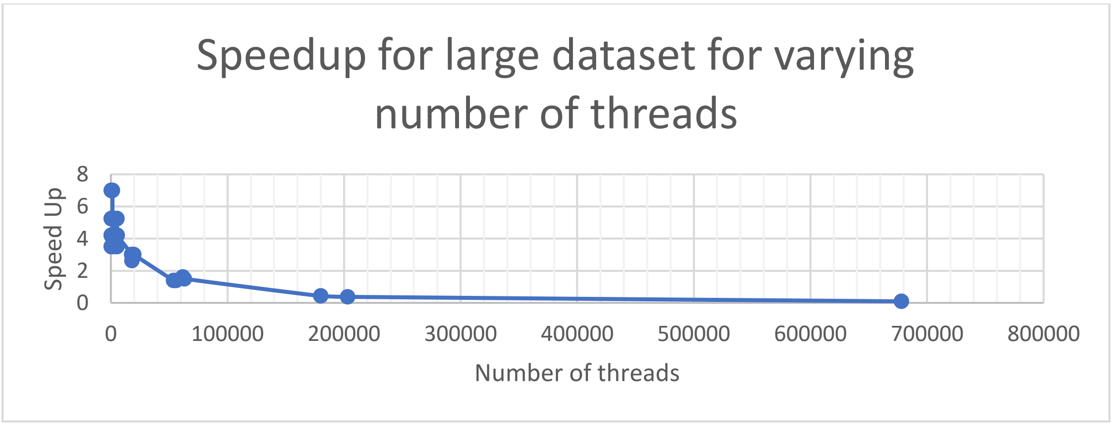

Figure 11: Speed Up for large dataset for varying number of threads

From the plots above, the below table was obtained that summarizes the optimal number of threads for each dataset for my system architecture, the optimal number of threads was around 4 000 as anything after that would lead to a diminishing return on the performance.

3.3 Comparison with Theoretical Speed Up

As discussed earlier, with respect to the algorithm applied.

$$Speed\ Up = \frac{n}{\log(n)}$$

From this, the table below compares the theoretical speed up to the actual obtained.

Table 4: Comparison of Theoretical vs Actual Speed Up

Dataset Size	Theoretical Speed Up	Max Obtained Speed Up
Small	13 600	4
Medium	48 400	6
Large	174 000	5

As noted in the above table, the obtained speed ups are significantly smaller than the expected ones. A possible reason for this lies in the limitation of the sequential block section. Since the comparison of the 8 neighbours is still a bit demanding, its likely a good potion of the time is spent there leading to the above observation.

4 Conclusions

4.1 It is worth using Parallelization to solve the basin identification problem

From the tested datasets, it was noted that although the obtained speed ups were not as large as the theoretical ones, they were still significant as we obtained something between 4 to 6 times speed up for datasets and if we are to scale up the problem to even larger datasets, this increase would be quite beneficial.

4.2 The parallel solution works best for medium datasets followed by large datasets

When assessing the speed up for the various datasets, it was observed that it performed the best for the medium dataset followed by the large dataset. However, since the large dataset was significantly larger, we could say that they both performed equally since there is only a difference of 1. At the same time, the same could be said about the small dataset.

4.3 For tested system architecture, you must ensure that threads are not more than 4000 for optimum speed Up

For this experiment, it was observed that the system architecture can not provide sufficient speed up for more than 4 000 threads and as such, when solving problems, it is best to stick to try to solve problems within this range. This has been observed to be also one of the reasons why the solution worked best for the medium dataset.