# MYNBRI003

## Assignment 3 Report

CSC2002S

Report on Parallel and Concurrent programming assignment 1

Brian Mynhardt
b.d.mynhardt@gmail.com

## Contents

## Introduction

The problem assigned to us was to for a canopy of trees and a set of sunlight values, to determine both the total sunlight exposure for each tree as well as the mean sunlight exposure for all trees within the area.

In this assignment we were tasked with parallelizing this sequential problem in such a way that it is sped up. We were to then evaluate the program experimentally by timing the run times of the serial and parallel methods, across a range of different input sizes. This as well as experimenting with different sequential cut-off parameters in order to see when the sequential processing should begin.

The Java Fork/Join framework was used to parallelize the problem of summing values of smaller grids contained within a larger grid. Splitting the array and then processing them within individual threads. Having run the program on a 4 Core 8 Thread architecture I expected a speedup of roughly 50% according to Amdahl’s Law.

## Methods

I approached the problem by first creating a sequential algorithm first as to see the base speed of that the machine could process the data at. This sequential algorithm works by simply going through an ArrayList of tree objects and for each one sequentially going through their contained values and summing them.

The parallelized version uses a recursive method and the Java Fork/Join frame work to compute the values. Again, the program first takes all the inputs and Instantiates the different variables such as the Grid Size and the various sunlight values for each position in the grid.

Then an ArrayList is populated with Tree objects. Each tree object has its own variables such its X position, Y position and the extent of its square canopy. The Tree class has one important method which is the getSunlight() method. This method takes in a 2D-float array as well as the grid’s sizes. The method iterates through the squares contained within the tree and adds them to the tree’s individual sunlight value before returning the total sunlight for the tree.

The recursive algorithm itself works by splitting up the array into smaller, more manageable parts that it can process on separate cores in other threads. This is done by taking in a maximum and minimum value. The minimum value is initials set to 0 and the max set to the array’s length. The method then recursively splits the array in half by setting a left side and right side to each half of the initial array. Once the difference between the max and the min is less than a specified cut-off number, the methods reverts to a sequential algorithm, summing the values for the trees. This summed value is then returned which in turn works back through the algorithm to the initial split where both the sides are added together to give the final summed value.

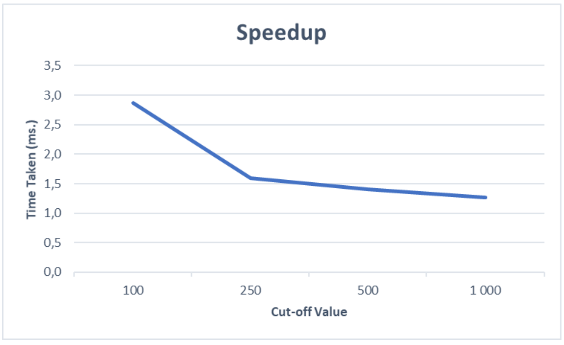
The final sequential cut-off value I used was 1000 after testing for different values first and seeing that 1000 gives the best overall speedup for my algorithms.

I timed the runtimes for the different algorithms by using the System tool: System.currentTimeMillis(). The time was taken only over the method which sums the values. I ran twenty summations for each dataset to get the times.

## Results

The first thing I tested the timing for was the different cut-off values for the parallel algorithm which yielded the result that on average, a sequential cut-off of 1000 works best for my program as can be seen from the table and diagram below.

| | Cut-Off Values | | |
|---|---|---|---|
| 100 | 250 | 500 | 1 000 |
| 8,9991 | 4,9995 | 0,9999 | 2,9997 |
| 1,9998 | 0,9999 | 1,9998 | 1,9998 |
| 0,0000 | 0,9999 | 0,9999 | 0,9999 |
| 0,9999 | 0,9999 | 0,9999 | 0,9999 |
| 0,9999 | 0,9999 | 0,9990 | 0,9999 |
| 10,0000 | 5,0000 | 2,0000 | 1,0000 |
| 2,0000 | 1,0000 | 1,0000 | 4,0000 |
| 1,0000 | 1,0000 | 2,0000 | 1,0000 |
| 1,0000 | 1,0000 | 1,0000 | 1,0000 |
| 1,0000 | 1,0000 | 0,9999 | 1,0000 |
| 9,9970 | 1,9994 | 3,9988 | 1,9994 |
| 2,9991 | 2,9991 | 0,9999 | 0,0000 |
| 0,0000 | 0,0000 | 0,9997 | 0,0000 |
| 0,9997 | 0,9997 | 0,9999 | 0,0000 |
| 0,9997 | 0,0000 | 0,9997 | 0,9997 |
| 2866,2801 | 1599,8200 | 1399,7600 | 1266,5533 |



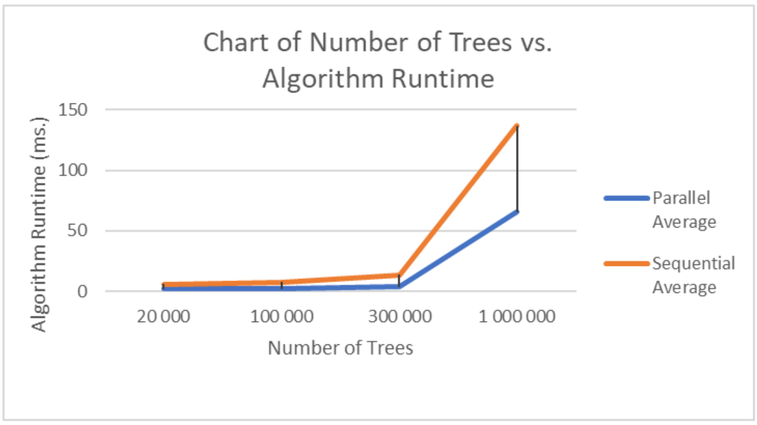
When I had that, I decided to run the sequential algorithms next and times them at various data sizes which produced the following results:

| | Sequential | | |
|---|---|---|---|
| 20 000 | 100 000 | 300 000 | 1 000 000 |
| 4,99950 | 7,99920 | 11,99880 | 1112,88870 |
| 5,99940 | 10,99890 | 13,99860 | 94,99051 |
| 3,99960 | 10,99890 | 12,99870 | 79,99200 |
| 4,99950 | 6,99930 | 11,99880 | 76,99230 |
| 4,99950 | 9,99900 | 12,99870 | 82,99170 |
| 4,99950 | 8,99910 | 12,99870 | 77,99220 |
| 4,99950 | 3,99960 | 12,99870 | 95,99040 |
| 3,99960 | 4,99950 | 13,99860 | 73,99260 |
| 3,99960 | 6,99930 | 13,99860 | 73,99260 |
| 3,99960 | 6,99930 | 19,99800 | 73,99260 |
| 4,99950 | 3,99960 | 17,99820 | 102,98970 |
| 5,99940 | 4,99950 | 11,99880 | 95,99040 |
| 4,99950 | 5,99940 | 11,99880 | 90,99090 |
| 4,99950 | 16,99830 | 12,99870 | 102,98970 |
| 9,99900 | 7,99920 | 12,99870 | 83,99160 |
| 8,99910 | 7,99920 | 12,99870 | 82,99170 |
| 8,99910 | 4,99950 | 12,99870 | 92,99070 |
| 5,99940 | 7,99920 | 11,99880 | 86,99130 |
| 5,99940 | 4,99950 | 12,99870 | 77,99220 |
| 4,99950 | 5,99940 | 12,99870 | 76,99230 |

The parallel algorithm was next after having settled on 1000 for the sequential cut-off value; The program then gave us a list of outputs like this:

| | Parallel | | |
|---|---|---|---|
| 20 000 | 100 000 | 300 000 | 1 000 000 |
| 3,99880 | 2,99910 | 3,99880 | 391,88245 |
| 0,99970 | 0,99970 | 2,99910 | 57,982605 |
| 1,99940 | 3,99970 | 2,99910 | 52,984104 |
| 1,99940 | 1,99940 | 3,99880 | 43,986805 |
| 0,99970 | 1,99970 | 2,99910 | 42,987105 |
| 0,99970 | 2,99970 | 2,99910 | 49,985006 |
| 1,99940 | 1,99970 | 2,99910 | 51,984407 |
| 0,99970 | 0,99970 | 2,99910 | 45,986205 |
| 1,99940 | 2,99970 | 2,99910 | 45,986205 |
| 0,99970 | 0,99970 | 3,99880 | 46,985906 |
| 1,99940 | 2,29970 | 3,99880 | 46,985906 |
| 0,99970 | 0,99970 | 2,99910 | 46,985906 |
| 1,99940 | 2,99970 | 3,99880 | 49,985006 |
| 0,99970 | 2,99970 | 3,99880 | 43,986805 |
| 1,99940 | 1,99970 | 4,99850 | 45,986205 |
| 0,99970 | 1,99970 | 2,99910 | 44,986505 |
| 0,99970 | 1,99970 | 2,99910 | 48,985306 |
| 2,99910 | 1,99940 | 5,99820 | 45,986205 |
| 2,99910 | 0,99970 | 2,99910 | 45,986205 |
| 2,99910 | 1,99940 | 4,99850 | 62,98111 |

When comparing the averages of the two algorithms as seen below, we can see that we have a speedup of roughly of around 4 times while still increasing as we increase the size of the datasets, However, there is a decrease in the speedup from around 600 000, as we see that there is only a speedup of around 200% for the dataset of 1 000 000 Trees



| | Parallel Average | Sequential Average |
|---|---|---|
| 20 000 | 1,79946 | 5,64944 |
| 100 000 | 2,11463 | 7,54925 |
| 300 000 | 3,64891 | 13,49865 |
| 1 000 000 | 65,68030 | 136,93631 |

## Discussion:

For this assignment I believe we can safely state that it is worth parallelizing this program as we get a speedup of at least 300%. I also believe that the process could possible be reduced to even smaller timeframes with a better algorithm.

The program behaves at its peak with a sequential cut-off around 1000 and with datasets that have between 200 000 trees and 500 000 trees. The best sequential cut-off value also increases as the dataset size increases until it reaches around 600 000, Where the speedup drops.

A useful observation made was that the amount of time taken to read in the data takes an extremely vastly larger amount of time compared to the actual summation and as such other input or database storage methods should be used in the future.

## Conclusions

From the data that I have collected I can see that parallelism can greatly increase the speed at which a computer processes data granted it has the ability to do so by having functionality for threads which most computers do in today's age. There are however pitfalls as unnecessarily parallelizing a program can also drastically reduce the processing speed due to various issues.

The data I have collected is accurate to 5 decimals so I believe that It is accurate enough. It shows that when switching over from a Sequential Summation algorithm to a parallel one, one can achieve speedup of over 3.5x. This is a pretty significant increase, but it can however be increased even more with a more efficient algorithm I believe.