Parallel Programming with the Java Fork/Join framework: Photosynthesis

Introduction

One of the most important factors affecting a tree's growth is its exposure to sunlight. Hence the ***aim of this experiment*** is to calculate the average sunlight received by trees in a forest (and also record the sunlight exposure of individual trees).

The Fork/Join framework is used to parallelize the sunlight exposure calculations using a divide-and-conquer algorithm. This approach is expected to speed up the calculations hence improving the overall performance compared to sequential programming.

Methods

The idea in this practical is very simple. The larger task can be divided into smaller tasks whose solutions are then be combined. This smaller tasks must be independent so that they can be executed in parallel.

Testing for Correctness of the algorithm

Step 01

Before I began writing a program to parallelize the sunlight exposure calculations using a divide-and-conquer algorithm. I wrote a sequential program first that I tested with different sets of data to verify that it is correct. Because of the large datasets involved. I decided to be more creative and created an algorithm that compared the output of the sequential file with a file containing the expected output. This was done for small files until I was sure the program was working correctly which I then verified by comparing the sample output with the sample output file given on Vula.

Step 02

Using the resources provided on Vula I then wrote the program to parallelize the sunlight exposure calculations using a divide-and-conquer algorithm. To test if this program was working properly, the output produced had to correspond with the outputs from the sequential file. This was done automatically by the compare files program that was indicated above***. After testing this program for different files and the output proved to be correct, I then concluded that the program is correct.*** At this stage I was only concerned about how correct it is and not the performance.

Timing Algorithms with different input and measuring the speedup

I generated 10 files of input data. The first file had 100,000 per tree data and the rest with increments of 100,000. The runtime of the data in each file was done 10 times using the ***System.currentTimeMillis*** method to do the measurements.

To make my work easier and avoid the inconveniences that may come with creating new files each time. I modified my program to only read the data that I have specified. Thus by changing the Terrain values and the number of trees, the program will read data only up to that section as if it were a new file. For every dataset, the 10 values were recorded in a table as shown below.

Sequential Cut off set at 50,000

Parallel program			Sequential program	
Dataset 100, 000				
Runtime 00	0,037		Runtime 00	0,321
Runtime 01	0,012		Runtime 01	0,204
Runtime 02	0,013		Runtime 02	0,206
Runtime 03	0,019		Runtime 03	0,175
Runtime 04	0,016		Runtime 04	0,178
Runtime 05	0,018		Runtime 05	0,194
Runtime 06	0,011		Runtime 06	0,185
Runtime 07	0,012		Runtime 07	0,203
Runtime 08	0,009		Runtime 08	0,19
Runtime 09	0,008		Runtime 09	0,219
Average	**0,0155**		**Average**	**0,2075**

Sequential Cut off set at 100,000

Parallel program program			Sequential Program	
Dataset 200,000				
Runtime 00	0,045		Runtime 00	0,441
Runtime 01	0,02		Runtime 01	0,409
Runtime 02	0,064		Runtime 02	0,364
Runtime 03	0,034		Runtime 03	0,356
Runtime 04	0,023		Runtime 04	0,365
Runtime 05	0,021		Runtime 05	0,46
Runtime 06	0,022		Runtime 06	0,406
Runtime 07	0,018		Runtime 07	0,357
Runtime 08	0,021		Runtime 08	0,372
Runtime 09	0,02		Runtime 09	0,35
Average	**0,0288**		**Average**	**0,388**

Sequential Cut off set at 200,000

Parallel program program			Sequential Program	
Dataset 400,000				
Runtime 00	0,062		Runtime 00	0,886
Runtime 01	0,065		Runtime 01	0,897
Runtime 02	0,032		Runtime 02	0,846
Runtime 03	0,03		Runtime 03	0,917
Runtime 04	0,032		Runtime 04	0,846
Runtime 05	0,031		Runtime 05	0,83
Runtime 06	0,033		Runtime 06	0,829
Runtime 07	0,032		Runtime 07	0,828
Runtime 08	0,031		Runtime 08	0,834
Runtime 09	0,032		Runtime 09	0,822
Average	**0,038**		**Average**	**0,8535**

Sequential Cut off set at 200,000

Parallel program program			Sequential Program	
Dataset 600,000				
Runtime 00	0,07		Runtime 00	1,169
Runtime 01	0,071		Runtime 01	1,225
Runtime 02	0,041		Runtime 02	1,157
Runtime 03	0,042		Runtime 03	1,159
Runtime 04	0,039		Runtime 04	1,208
Runtime 05	0,042		Runtime 05	1,18
Runtime 06	0,041		Runtime 06	1,165
Runtime 07	0,042		Runtime 07	1,227
Runtime 08	0,04		Runtime 08	1,209
Runtime 09	0,072		Runtime 09	1,161
Average	**0,05**		**Average**	**1,186**

Sequential Cut off set at 500,000

Parallel program program			Sequential Program	
Dataset 800,000				
Runtime 00	0,086		Runtime 00	1,504
Runtime 01	0,105		Runtime 01	1,499
Runtime 02	0,084		Runtime 02	1,573
Runtime 03	0,049		Runtime 03	1,516
Runtime 04	0,054		Runtime 04	1,513
Runtime 05	0,049		Runtime 05	1,535
Runtime 06	0,047		Runtime 06	1,519
Runtime 07	0,056		Runtime 07	1,569
Runtime 08	0,054		Runtime 08	1,551
Runtime 09	0,056		Runtime 09	1,507
Average	**0,064**		**Average**	**1,5286**

Sequential Cut off set at 550,000

Parallel program program			Sequential Program	
Dataset 1000,000				
Runtime 00	0,128		Runtime 00	1,876
Runtime 01	0,124		Runtime 01	1,87
Runtime 02	0,063		Runtime 02	1,934
Runtime 03	0,067		Runtime 03	1,904
Runtime 04	0,067		Runtime 04	1,89
Runtime 05	0,067		Runtime 05	1,865
Runtime 06	0,067		Runtime 06	1,891
Runtime 07	0,065		Runtime 07	1,878
Runtime 08	0,066		Runtime 08	1,913
Runtime 09	0,093		Runtime 09	1,869
Average	**0,0807**		**Average**	**1,889**

There are many results that were collected not indicated here. However, it was determined that the program speeds up with reduction in dataset.

Maintaining the large value of the dataset the program achieves almost an optimal performance at 125000 which is basically the dataset size/ 8

Machine Architectures

The program was tested in two types of machines. Lenovo core i7 and Lenovo core i5. The performance in both machines was the same and I was not able to note any differences hence the data got was just similar to the one presented above.

Problems encountered

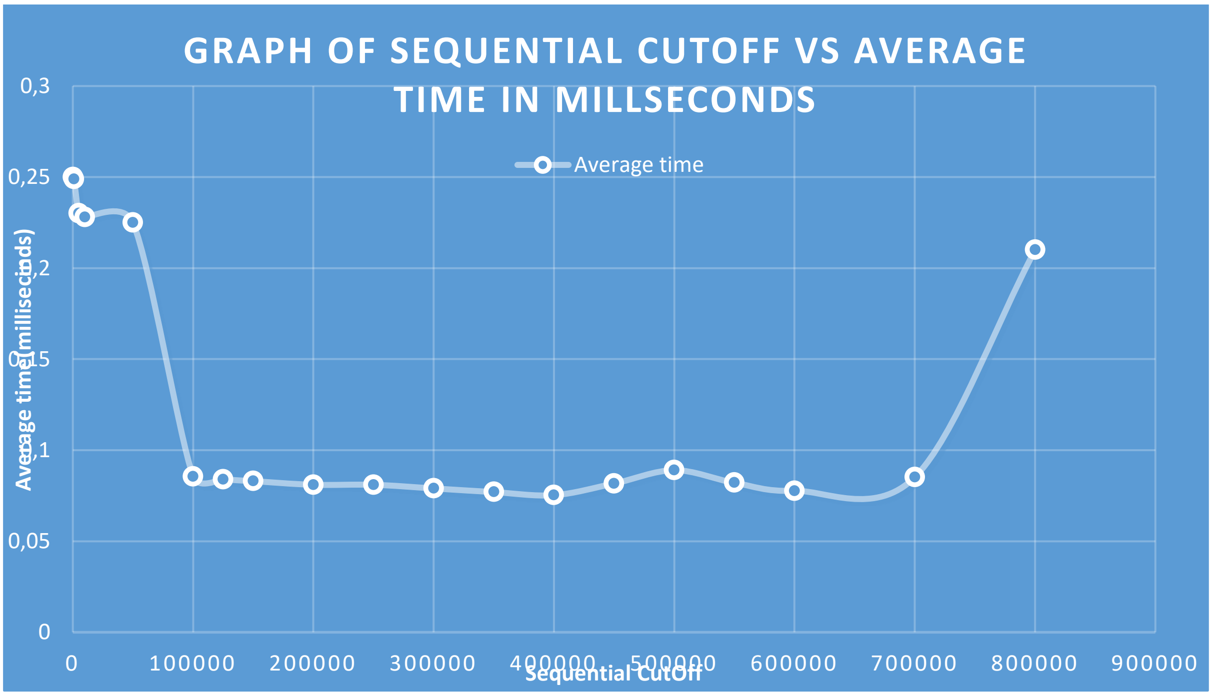
During the experiment I encountered the following problems.

- The sequential code I developed was not suitable for parallelization and hence I had to modify it for some time.
- During the testing phase for correctness, it was sometimes hard to tell if it is really parallelizing the data since it was small hence the time difference too small to tell
- Working with the large sets of data was difficult considering the machines that we use. The editors were not able to open them hence I had to rely on the terminal.

Results and Discussion

Sequential Cu off vs Average Time

The graph below was drawn based on the time the program takes to do the calculations for a fixed amount of data #1000, 000 dataset but with different sequential cut-off values



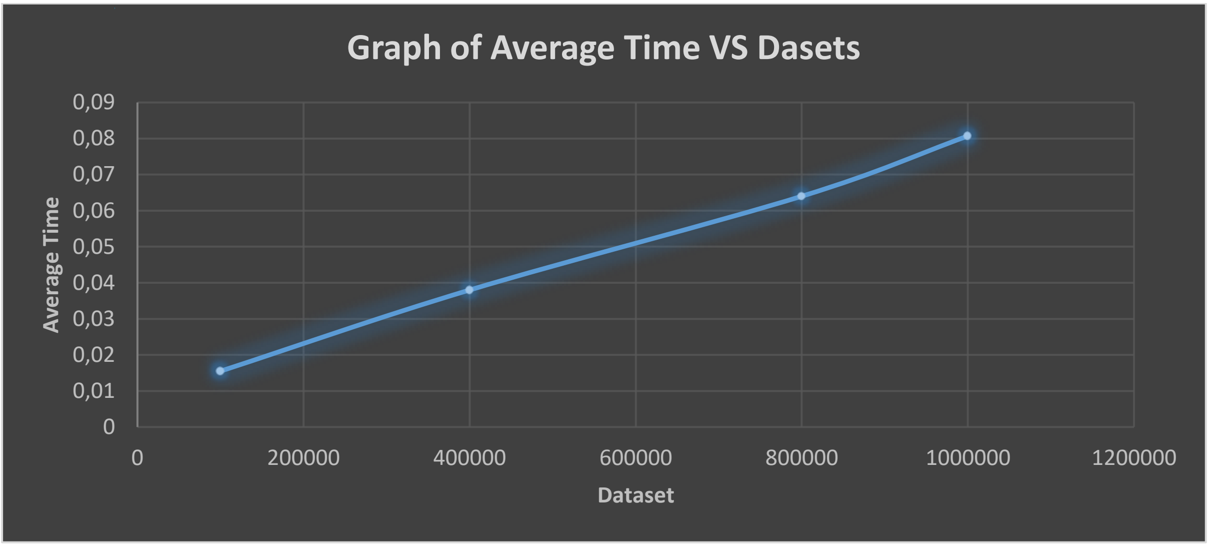
From the graph above the following was determined.

- The parallel program achieves highest performance for the 1000,000 dataset when the sequential cut-off value is set at around 600, 000. However based on the research which I did on the internet. My expectation was 550,000. Since this is a parallel program, I believe

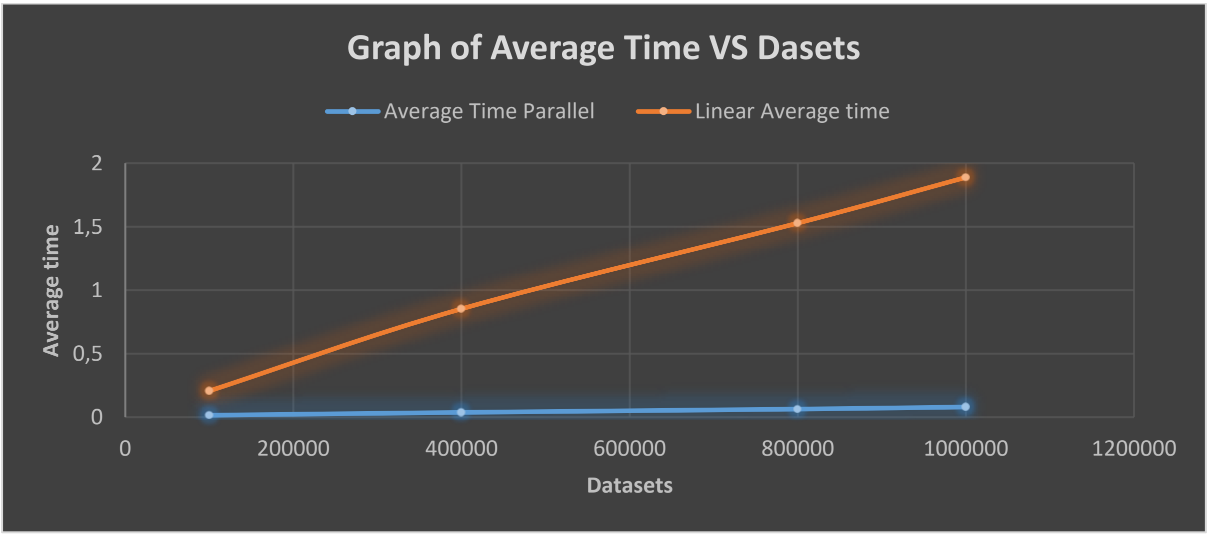
550,000 and 600,000 are not very different. Hence I do believe the program is working as expected and with the expected performance.

Data sizes vs Average time

The graph below shows how the increase in number of datasets affects the run time of the parallel program



From the graph above it can be seen that the average time taken to perform the calculations increases with the amount of datasets though it is very minimal. The differences between the linear time and the time the parallel program takes can be identified in the graph below.



Based on the above results***, it can be seen that, using multithreading is very worth it as it helps speed up the performance of the calculations unlike the sequential code*** which is very linear and increases with increase in dataset.

Observing the above three graphs we can also see that the ***program performs very well in data ranging from 125, 000 onwards.*** I would not make a conclusion on where the program might become ineffective as I didn't surpass the 1000,000 dataset.

Conclusions

Based on the above results I would like to make the following conclusion.

- The aim of this projective to parallelize the sunlight exposure calculations using a divide-and-conquer algorithm to speed up the calculations has been achieved
- The parallel programs are only efficient for large sets of data.
- The Sequential program is very efficient for sets of data below 600,000 taking the case of this problem
- The sequential cut-off for this problem is 600,000
- Use large sets of data to test for correctness of parallel programs.