THNGEO002

Parallel vs. Sequential Programming

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

Georgeo Thanathara
9/4/2019

Table of Contents

Introduction

Aim

The aim of this project is to compare the speed up times of using the java fork/join Framework against using a regular sequential program. I am going to show how using parallel programming in this manner will provide correct and faster results than a serial program, this is done by performing experiments on the outputs of a cloud simulation data, in which the prevailing wind averages are calculated and the cloud classifications are assigned.

Parallel Algorithm

Parallel algorithms allow instructions to be processed simultaneously on devices that have more than one core; it makes use of the available CPU cores and returns a result at a quicker time than using a serial program.

In this experiment, the java Fork-Join framework is used. This framework is designed to meet the needs of the divide and conquer fork-join parallelism by making use of the java.util.concurrent package. This framework supports the use of light weight threads and is small enough such that millions of the threads can be created.

The divide and conquer algorithm ensures the accumulation of partial results across the threads by making recursive calls (breaking down the problems into smaller sub problems) while trying to access all the available processor cores.

Expected speed ups

Using this algorithm the total time is the tree height which is O(log n) , this is exponentially faster than using a serial approach which would be O(n).

With relation to the sequential cutoff, in theory, using the divide and conquer to bifurcate to a single element will ensure optimal speed of O(n/p +log(n)) but in practice, the creation of the multiple threads and communication between them will swamp the savings, eventually increasing the time.

We use Amdahl's law to give the theoretical speed up of the implementation of the parallel program.

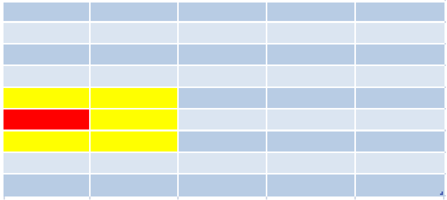
The speed up formula is $\frac{1}{(1-p)+\frac{p}{n}}$ where p is the parallel fraction, n is the number of processors and (1-p) is the sequential fraction. This formula shows that speed up is dependent on the resources of the system although it contains flaws within it.

Method

A cloud simulation data was provided in order to conduct this experiment. With the provided data, the prevailing wind averages and the cloud classifications are calculated. The prevailing wind averages are calculated by getting the average x and y values of all the data in the simulation. The cloud classifications are determined by getting the local averages and comparing it to the uplift value of the current element in the matrix (a particular cloud in the simulation).

Local average & classification

The local average is the average of all the neighboring elements as well as the element itself. This is illustrated in (figure a).



(figure a)

For eg. If we want to find the local average of the red element in the 9x5 matrix, we calculate the average of all the yellow elements and the red element to get the local average of that specific red element, hence why it's the average of all the "neighboring" elements

We are going to use (ixj) to define each element on the matrix

To implement this in code, I ensured that all possible neighbors are used. This consisted of 9 IF statements, one for each element you are on. There can only be a total of 8 neighbors in the worst case scenario, and that is when you are on the "middle" element. For the corner cases I checked if a) i=0,j=0 b)i+max,j+max c)i=0,j+max d)i+max,j+0. Finally, for the non corner cases and non middle case I checked for a)x=0 b)y=0 c)i=max d)j=max. This ensures that all neighboring cases are dealt with and produces the correct local average for each element.

The classification of each element was assigned immediately after the local average was calculated. The classification was determined by comparing the uplift value and wind magnitude (which is calculated using the local average x and y values using the formula $\sqrt{x^2 + y^2}$). The given conditions to compare the magnitude and uplift were implemented and a classification (0, 1, and 2) was assigned to each element in the matrix.

The serial method

sequentialCalc.java

The serial method was simple to implement. It first takes in the file to be read and the file to be written to, which then makes a call to a method called average(fileR,fileW) passing the two file names as parameters. A vector wind is then created to store the corresponding x and y values of the prevailing wind average and an object of the *provided* CloudData.java is created. Three for loops are used in order to get to each element in the matrix, upon which the x and y values of each element are added to a cumulative total respectively and a call to the local average and classification is made in order to determine the classification and assign the value to it. The prevailing wind average is calculated by dividing the cumulative total by the dim method in the CloudData. The output is then written to the file using the write method of the cloud data.

The parallel method

Parallel.java parallelCalc.java

The parallelCalc class fundamentally makes use of the concurrent package and specifically uses the ForkJoinPool framework as mentioned above. In the main, the file is read, a wind vector object is created and a call to the invokeP(CloudData obj,int size) method is made. The invokeP method takes in the CloudData object and the size of the data, which then calls the invoke method (while sending the CloudData object, 0 and the size as parameters to the constructor of the Parallel class) of the object created from the ForkJoinPool, This method is used to perform a task that is specified and returns an answer. In this case, a vector is returned for the prevailing wind averages.

The parallel class extends a subclass from the ForkJoinPool known as RecrsiveTask; this is an abstract class that returns a value to you. This is particularly useful in this situation as it uses the reduction pattern to produce a single answer. This class uses a generic data type to return, thus in this case the return type is a Vector.

The compute method uses recursion in order to split each task into smaller subtasks following the divide and conquer method. We use a sequential cutoff (to reduce the amount of swamps) as the base case in the recursive function, once the tasks are split into smaller subtasks and the sequential cutoff is met (calculates by high minus low) the program then performs the sequential part of the program. This uses the locate method from the CloudData class to trace through each element to assign the prevailing wind x and y vector, as well as set its classification. The right subtask, uses fork to split up the process in threads while the left subtask only uses the compute such that it reduces the number of threads. The join is then used to ensure that the execution of the threads is completed. The returned vectors from the left and right subtask are then added and returned back to the parallelCalc class as a vector.

Authenticity of the Results

I ensured that my results were correct by two methods:

1. Using the terminal with the command *diff largesample_output.txt testOutput.txt > error.txt*. This gives me any item that do not match between the two files in another file called errors.txt
2. Using the website https://diffchecker.com/diff. This website allows me to put the contents of two files and returns and shows which parts of the file are not the same by highlighting in red.

After applying both methods to my output, It was guaranteed that were no errors in the output. The results I was getting were correct.

Timing and speed up

To be able to have accurate timing of the execution of the serial and parallel program, I used currentTimeMillies which is used to return the current time in milliseconds. tick() is a method which records the starting time of the stopwatch and tock() gives the time it took to run the program by subtracting the current time from the start time.

Because the java virtual machines have not optimized the library internals, the time calculated for the first few tests are not accurate enough. To prevent this erroneous time results, the computations are calculated 500 times in a loop and an average of the time is reported. This ensures the results that are obtained are correct and valid.

The speedup is determined by getting the ratio of the best serial times and the best parallel times(by best I mean the sequential cutoff that provided the fastest time). Speed up is noted for the different data sizes available.

The formulas for Sequential cut off used in the experiment is {xE$Z^{+}|10^{x}: x < 6$ } thus sequential cut offs of 10,100,1000,10000,100000 were used to ensure that a wide range of cutoffs can be used to determine the best optimal sequential cutoff for a given data size.

Architectures

1) Toshiba Windows 10 2013

Processor	Intel® Core™ i7-3630QM CPU @ 2.40GHz
Installed RAM	6.00GB
System type	64-bit operating system, x64- based processor
Cores	4
Logical Processors	8
Maximum Speed	2.40GHz

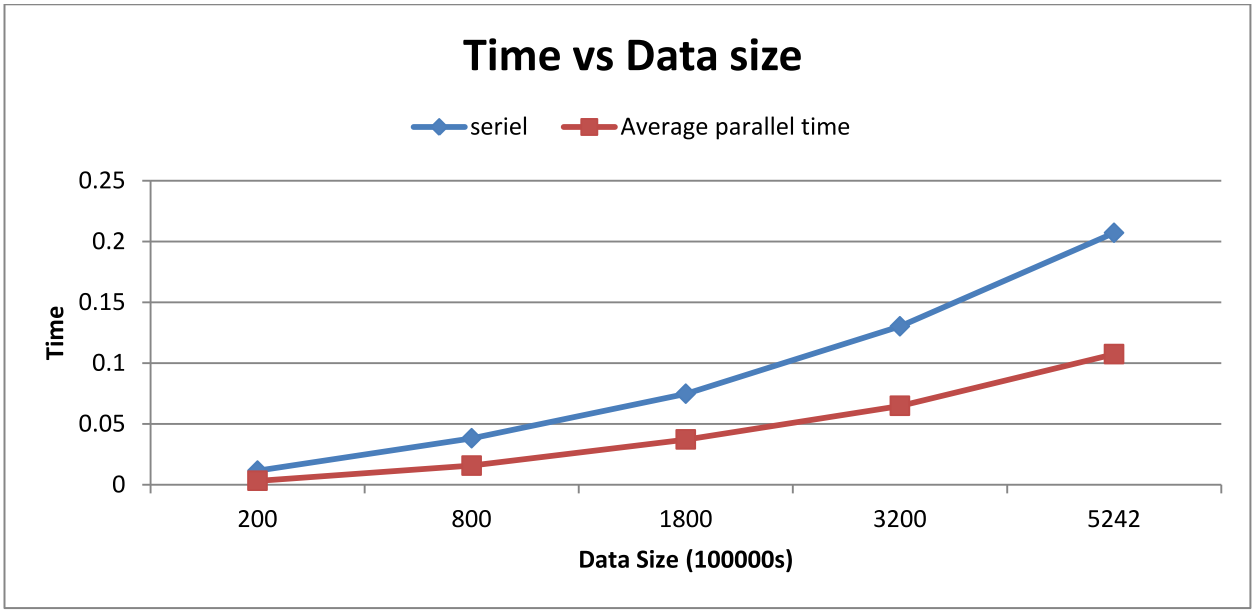
2) Toshiba Satelite Windows 8.1Pro 2016

Processor	Intel® Core™ i3-5005U CPU @ 2.00GHz
Installed RAM	4.00GB
System type	64-bit operating system, x64- based processor
Cores	2
Logical Processors	4
Maximum Speed	2.00GHz

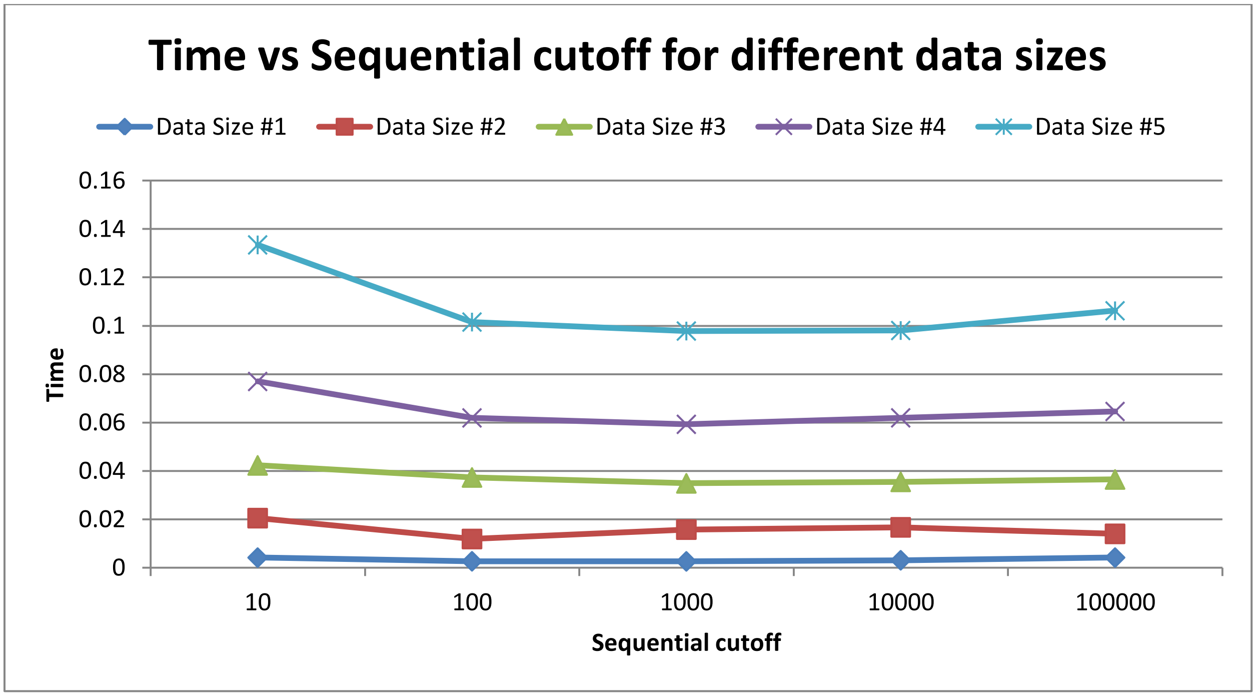
Results and Discussion

Results

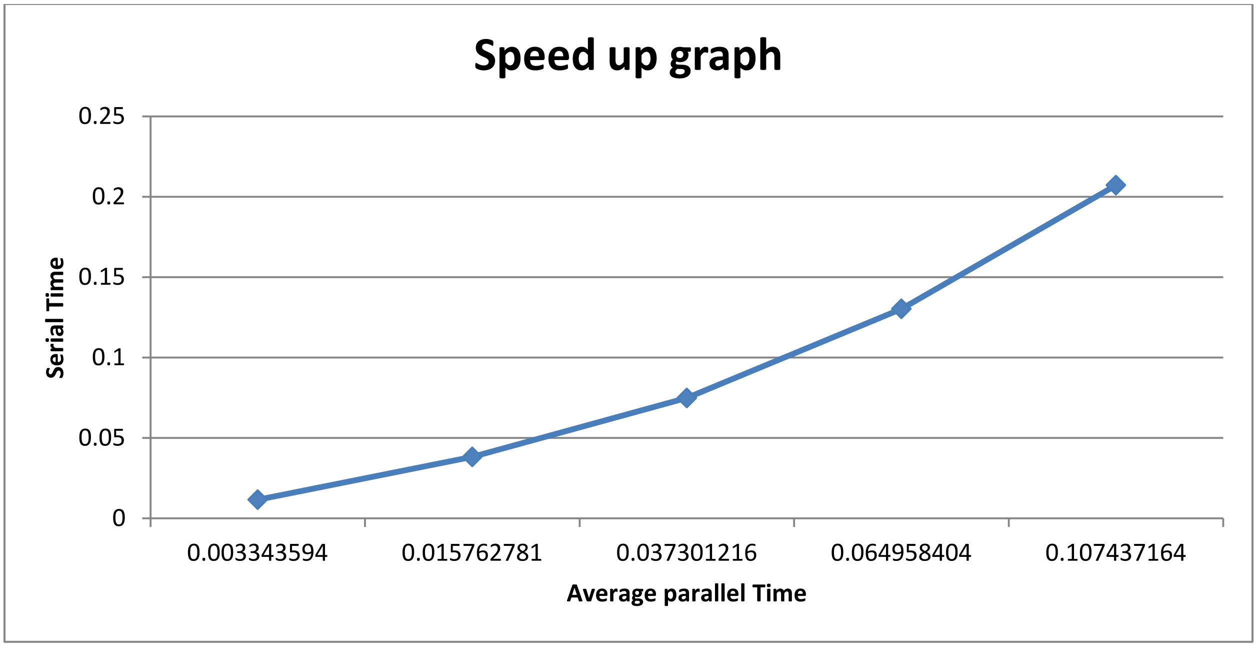
Architecture 1: Toshiba Windows 10



(Figure b) Time vs data size of serial and parallel.



(Figure c) Time vs Sequential cutoff for parallel



(Figure d) Speed up graph of Serial time against the Average Parallel Time

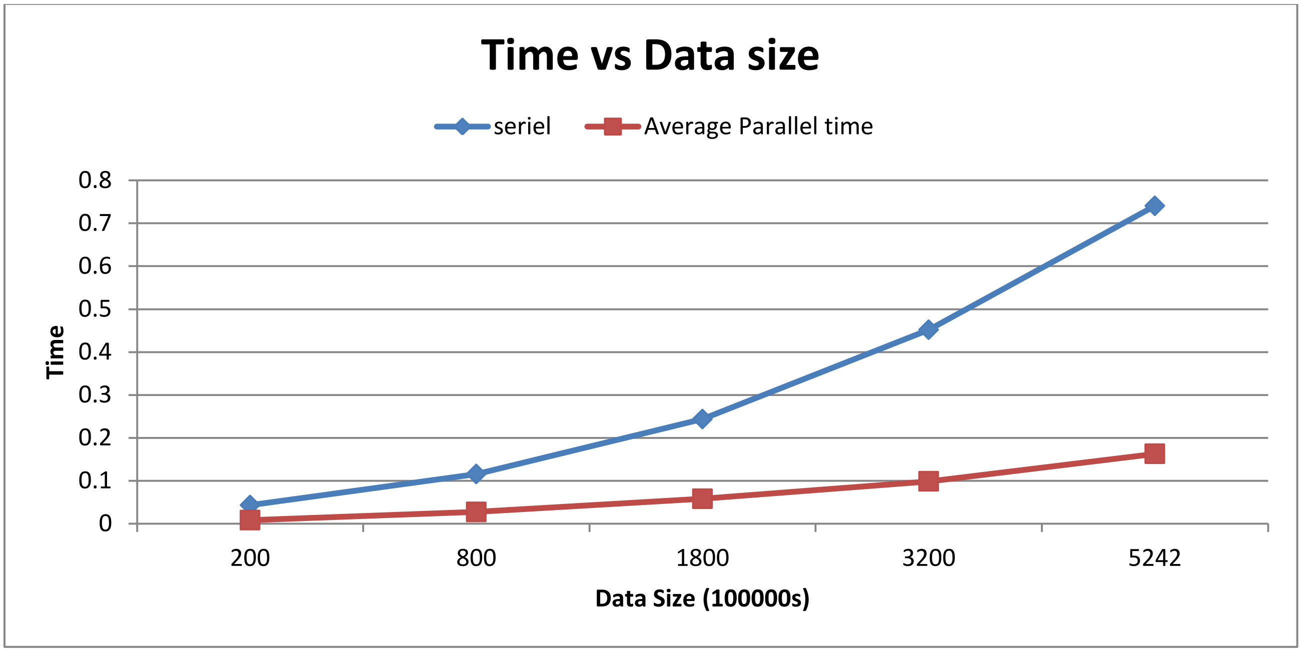
Table for the Parallel Times for different sequential cutoff and data sizes

Sequential Cutoff	Data Size #1	Data Size #2	Data Size #3	Data Size #4	Data Size #5
10	0.004174011	0.020491999	0.04230395	0.077001944	0.13346203
100	0.002657992	0.011867987	0.037320007	0.06193998	0.10156985
1000	0.002625992	0.015765952	0.03488606	0.059288118	0.09785002
10000	0.003069998	0.016685946	0.035498045	0.061959997	0.09807597
100000	0.004189976	0.014002022	0.03649802	0.06460198	0.10622795
Average	0.003343594	0.015762781	0.037301216	0.064958404	0.107437164

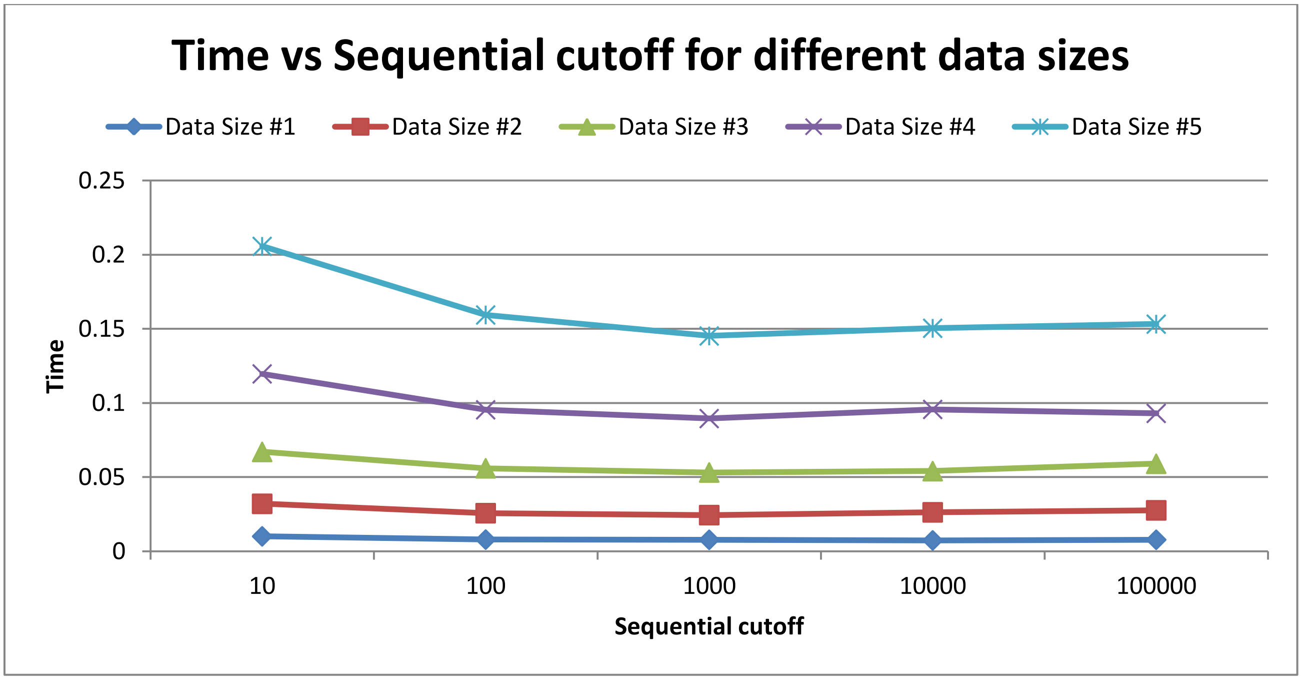
Table for serial times with different data sizes

Data Size	time
200	0.01165404
800	0.03825406
1800	0.07487812
3200	0.13031043
5242	0.20719464

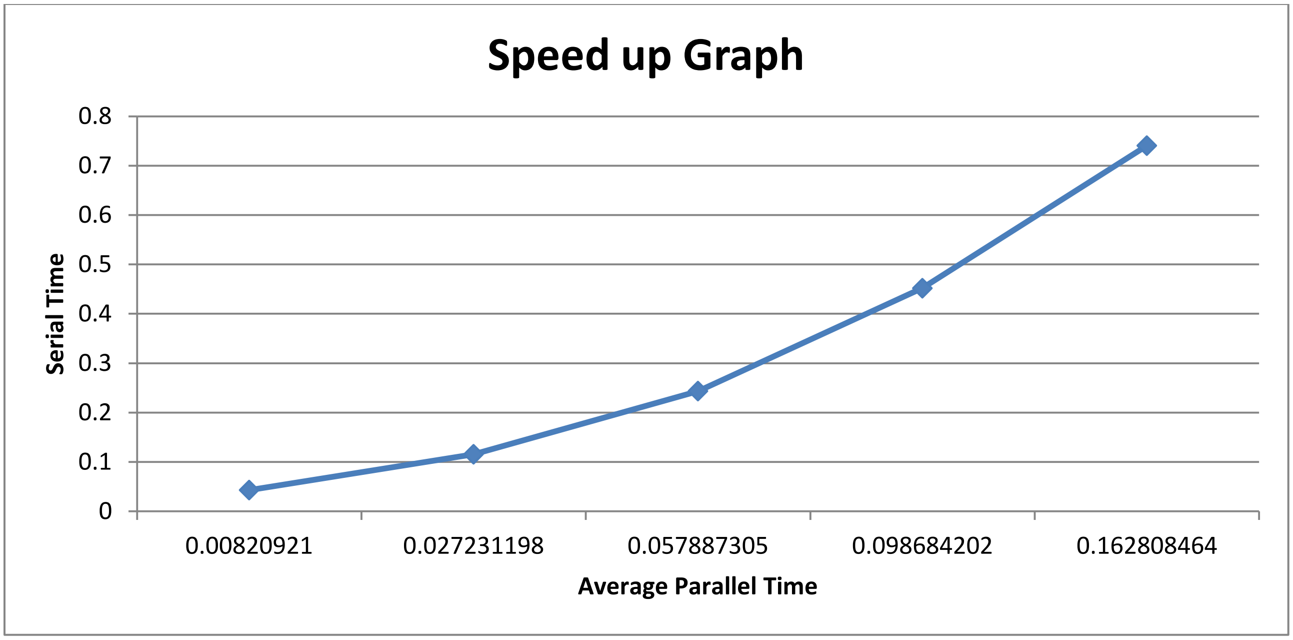
Architecture 2: Toshiba Satelite Windows 8.1Pro



(Figure e) Time vs data size of serial and parallel.



(Figure f) Time vs Sequential cutoff for parallel



(Figure g) Speed up graph of Serial time against the Average Parallel Time

Table for the Parallel Times for different sequential cutoff and data sizes

Sequential Cutoff	Data Size #1	Data Size #2	Data Size #3	Data Size #4	Data Size #5
10	0.010111992	0.032052014	0.06707394	0.11968421	0.20569666
100	0.007924016	0.025723996	0.056014147	0.09545225	0.15942143
1000	0.007802014	0.024431989	0.053192165	0.08954802	0.14526746
10000	0.007390013	0.026297996	0.054098178	0.0957023	0.1504274
100000	0.007818015	0.027649993	0.059058093	0.09303423	0.15322937
Average	0.00820921	0.027231198	0.057887305	0.098684202	0.162808464

Table for serial times with different data sizes

Data Size	time
200	0.043192048
800	0.115655966
1800	0.2436297
3200	0.4518955
5242	0.7405875

Discussion

Is it worth using parallelization (multithreading) to tackle this problem in Java?

As we look at our results (specifically figure b and figure e) we see that the use of multithreading in parallelization does in fact improve the time taken to execute and finish the program. For either architecture, we notice the time taken in parallel program is much faster than it would take in the serial program. Regardless of which architecture, it is visible to the eye that the serial program is increasing time exponentially compared to the parallel program as the data size increase.

Speed up is calculated as the ratio of serial time to parallel time, thus $\frac{Serial\ Time}{Average\ Parallel\ time}$, If we take the gradient of the graphs in figure d and g we can get the speed up as seen in the table below.

Data size	Architecture 1
200	3.485483299
800	2.426859798
1800	2.007390837
3200	2.006059607
5242	1.92851926

So yes, It is worth tackling this problem in parallel. Although, it will depend on the architecture you use to notice any increase in execution time. From the speed up graphs in figure d and g, we can see what the parallel equivalent of the serial time would take, and the results all favor parallel processing.

For what range of data set sizes does the parallel program perform well?

The data set sizes chosen for this experiment was 200000, 800000, 1800000, 3200000 and 5242000. To answer this question we are not going to considering specific sequential cutoffs for the parallel program results, instead, we are going to work with averages of the results. The table below shows the difference between the serial and average parallel times of execution for the different data sizes.

Data size	Serial - parallel
200	0.008310446
800	0.022491279
1800	0.037576904
3200	0.065352026
5242	0.099757476

(Architecture 1)

In the table above, we can see that parallel program performed better for all the data sizes that we have tested. But more importantly, you can notice the pattern at which the difference between the serial and the parallel program is actually increasing as the data sizes gets bigger; this shows us that for the serial program the time is increasing, but for the parallel program the time is actually decreasing hence giving a greater range. The parallel times used in the calculation in the table is an average parallel time from the different sequential cutoffs. Thus, for some sequential cutoffs the serial program will actually perform better, but when looking independently at data sizes, we can see that the parallel time improves as the data size gets better.

What is the maximum speedup obtainable with your parallel approach? How close is this speedup to the ideal expected?

For the data sizes that I used for testing in these experiments, the maximum speed up noticed was for the smallest data size (of the average parallel times) was 3.48. Speed up is the ratio of serial/parallel. Thus given the expected speed up, which was log(n) = log(200000) = 5.301. We notice that the practical speed up was not equal to the ideal speed up.

There was many reasons for this, let us go through some;

The number of cores in architecture one was 4 cores, Usually parallelism does not work to completeness with devices with cores. If we used a device with more cores (Like 8 or 16) we could've noticed exponential increase in execution time, this is because tasks that are smaller could be pipelined more quickly.

During the testing phases there could have been background tasks that were processing, thus not giving the fork join pool full access to all the cores. Inevitably, making time closer to a serial program.

Although the time for execution in parallel for any architecture was much more dominant that its serial program, the speed up was still not affected that much, this can be due to the above mentioned situation or other problems regarding the availability of cores and efficiency of the cores, which causes the serial time to take much longer for a bigger data size, but the parallel (was still more efficient but) a little slower than its full capacity.

What is an optimal sequential cutoff for this problem?

The most optimal sequential cutoff for the data sizes that were tested is 1000. Figure c and f shows you the time it took for a particular data size to execute on 6 different sequential cutoffs. I used a wide range of sequential cutoff in order to properly determine which sequential off would be most suitable for the data sizes.

After inspection from the graphs, a sequential cutoff of 1000 was the one that gave the lowest time for execution. We can see that in fact the most appropriate range is between 100 and 10000 with 1000 giving the lowest time for the given data sizes.

A sequential cutoff of below 100, shows how the time increases, and a sequential cutoff greater than 10000 shows how the time also slowly starts to increase as well. The results obtained here is specifically for the chosen data sizes. For eg, the smallest data size is the most obvious when looking at the graph, we see when the sequential cutoff gets closer to the to the data size, the time slowly starts to increase and will eventually take longer to run than the sequential program if the sequential cutoff exceeds the data size.

What is the optimal number of threads on each architecture?

Since we are using the divide and conquer algorithm, we can calculate the number of threads that is being used within each data size using the formula $\frac{Data\ Size}{Sequential\ Cutoff}$. Thus in order to get the optimal number of threads in each architecture, we are going to calculate the average data size (of all data sizes we have used) and divide it using the best sequential cutoff that we have obtained.

Average Data size (x10^3)= 200+800+1800+3200+5242= 2248.4

Thus number of threads = $\frac{2248400}{1000}$ = 2248.4 threads

Thus the optimal number of threads for architecture 1 was 2249. Due to the fact that the second architecture used had only 2 cores while architecture 1 had 4 cores, I noticed that the optimal sequential cutoff was the same for the experimented data sizes , and thus the optimal number of threads for architecture 2 would be the same.

Conclusion

For the given data sizes and the chosen sequential cutoff, we can conclude that the parallel program was much quicker than when we did it serially. Although, this comes with a few things to consider when performing experiments like this;

In this experiment, the work being done at the base level is actually quite small, which means it is actually more difficult to make the parallel program to perform much better than the serial one, hence the speed up is not a large amount, it differs by a single amount as the data size gets bigger. This means that the work at the smallest level is too small, hence it is more difficult to get better speed up.

The data sizes to consider should be extremely large sizes when trying to test the efficiency of parallel programming. We can see that the data sizes used in this experiment did lead to a significant improvement in time, but is not significant enough. One of the flaws in Amdahl's laws was that typically Tp(The span) grows faster than Ts(The work) when the data size increases, Which will show an increase in efficiency as the ratio of Ts to Tp is not constant.

The sequential cut offs used in this experiment showed the most appropriate sequential cutoff for a specific data size. The fork join Documentation from java states that a task will ideally execute between 1000 and 10000 computational steps. And the results that we achieved proved this to us. By using a wide range for the sequential cutoff we were able to see after what computation steps did the parallel program start performing better, and similarly, when the parallel program started to take longer to execute.

In conclusion, when we are trying to test the speed up/ Efficiency of parallel programming, we must carefully consider the specifications of the architecture that we are working in. We have already showed that parallel programming performs best when large data sizes are used but we notice that architectures with 2 cores or 4 cores do not give results that are extra ordinary. When working with parallel programming it is essentially to make use of all the cores within the architecture, spanning minimum overheads as possible.

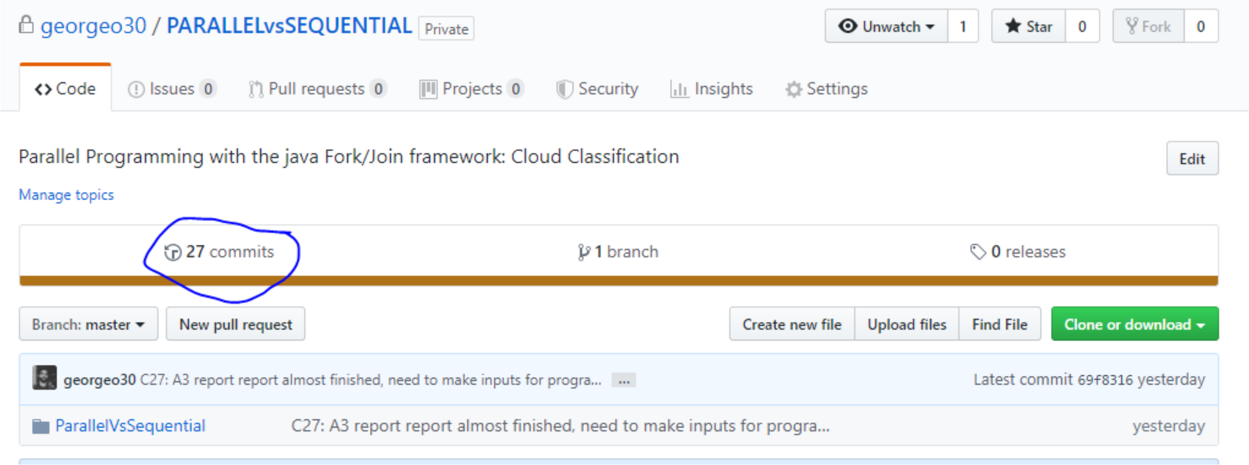
Git

Git repository can be found on my gitHub page : georgeo30 https://github.com/georgeo30

ParallelVsSequential repo : https://github.com/georgeo30/PARALLELvsSEQUENTIAL

If the above link does not work, Please go to my gitHub and search for ParallelvsSequential : It could be because the project is currently set to private at time of submission due to plagiarism purposes.

The project will be made public on the day of submission.



georgeo30 / PARALLELvsSEQUENTIAL Private

Parallel Programming with the java Fork/Join framework: Cloud Classification

georgeo30 C27: A3 report report almost finished, need to make inputs for progra... Latest commit 69f8316 yesterday

ParallelVsSequential	C27: A3 report report almost finished, need to make inputs for progra...	yesterday