MOHEE Bhavish
MHXBHA001

Assignment 3 Report

Introduction

In this project, the output of a simple weather simulation code will be analysed to determine the prevailing wind direction and predict the types of clouds that might result, based on values read from input text files. However, weather simulation is a rather complex task to be completed as it needs to be carried out within a specific time limit in order to maintain its importance and value. Hence, parallel programming will be implemented in this project in order to demonstrate its speed up against typical sequential programming.

Parallel programming is a rather recent form of computation in which execution of processes are done simultaneously by dividing large problems into smaller ones, which can be then processed at the same time. Parallel programming is excepted to provide a significant speedup in performance compared to sequential programming, depending on the input file size used and the sequential cut-off in the code.

Methods

As per my approach to parallelization of the weather simulation program, the main class has to import the *"java.util.concurrent.ForkJoinPool"* while the class where all the parallel processes occurs, namely *"MakeParallel"* class, has to import *"java.util.concurrent.RecursiveTask"* and also extends the *"RecursiveTask<Vector>"*.

The MakeParallel class initialises a defined final integer as the sequential cut-off and has a constructor which takes in 2 integers boundaries according to the amount of input data and a *CloudData* object. Afterwards, upon calling on the *compute* method by the main class, the whole parallelization process is initiated.

At first, by basic recursion, the data is divided into smaller portions until they fall within the pre-set sequential cut-off value. Then, the *left MakeParallel* object is allowed to run in parallel by calling upon the *fork* method whereas the *right MakeParallel* object is run sequentially in order to create further speed up. This process continues until the whole data has been computed.

- *Validating the algorithm*
  A fully sequential algorithm is also constructed in order to validate the correctness of its parallel version. Providing the same input text file to both algorithms should create the same output text file. Furthermore, since rather large output text files are being generated, some java codes can be computed into an algorithm which takes in both output files and compares them to try to find a difference. Being unable to spot one, actually validates the parallel program.

- *Timing the algorithm*
  The "*System.currentTimeMillis*" method is called upon and assigned to a double variable before the parallel section of the code. At the end of this particular section, the same method is called upon and subtracted from the previous double variable to result into the run time of the algorithm.

- *Measuring speed-up*
  The above-mentioned timing technique is applied to both the sequentially-programmed algorithm and the parallelly-programmed algorithm and the run-times are appropriately recorded in an excel table. Now, to demonstrate the speed-up visually, a graph of run-time against input file size is plotted for both programming techniques.

- *Machine architectures used*
  For this project, I have used 3 different machine architectures:
  1. HP Pavillion Laptop Intel® Core™ i7-8550U CPU @ 1.80GHz 1.99GHz
  2. CompSci Lab PC Intel® Core ™ i5-7400 CPU @ 3.00GHz x 4
  3. HP ProBook 450 G3 Notebook Intel® Core™ i5-6200 CPU @ 2.30GHz 2.40GHz

- *Problems/Difficulties encountered*
  The "*OutOfMemoryError:GC overhead limit exceed*" error was a problem I encountered whenever I tried using too large input text files (approximately 1 Gigabytes). So, I had to use input text files below that limit in order to prevent this error, hence, restricting my boundaries for testing parallel programming.

Results and Discussion

The tables below demonstrate the different input file sizes and run times in sequential format and parallel format on the 2 architectures used:

Please note that the following run time values were obtained by allowing the algorithm to run with the same input data a few times before actually recording run time.

Architecture 1 : HP Pavillion Laptop Intel® Core™ i7-8550U CPU @ 1.80GHz 1.99GHz

Input File		Sequential (seconds)	Parallel (seconds)					
			Sequential Cut-off					
Size(MB)	Dim()		100	500	1000	5000	10000	50000
2.784	100000	0.066	0.153	0.079	0.127	0.113	0.069	0.062
146.4	5242880	0.666	0.313	0.227	0.303	0.257	0.25	0.294
347.9	12500000	1.529	0.373	0.352	0.43	0.262	0.303	0.402
434.9	15625000	2.172	0.625	0.354	0.456	0.354	0.445	0.44
695.8	25000000	3.233	0.623	0.472	0.675	0.491	0.515	0.567

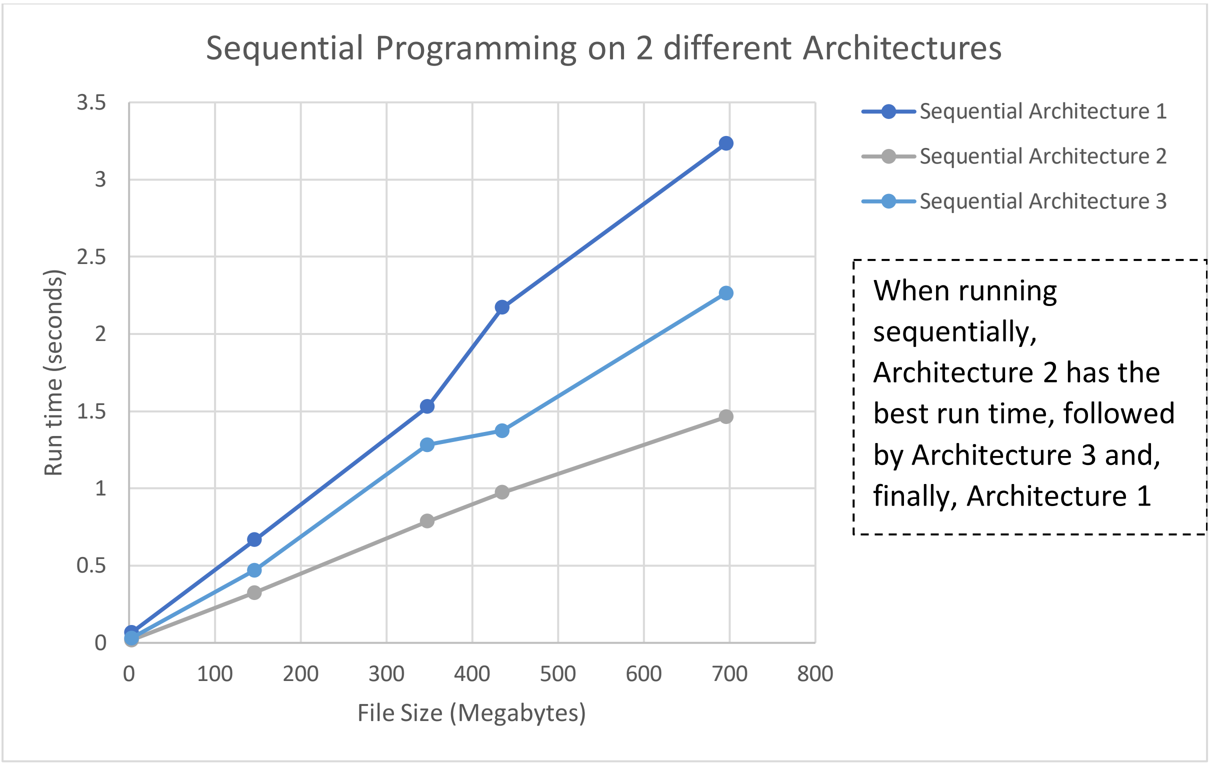
Architecture 2 : CompSci Lab PC Intel® Core™ i5-7400 CPU @3.00GHz x 4

Input File		Sequential (seconds)	Parallel (seconds)					
			Sequential Cut-off					
Size(MB)	Dim()		100	500	1000	5000	10000	50000
2.784	100000	0.018	0.087	0.076	0.077	0.074	0.06	0.072
146.4	5242880	0.324	0.4	0.24	0.269	0.201	0.201	0.268
347.9	12500000	0.787	0.521	0.38	0.446	0.432	0.403	0.401
434.9	15625000	0.973	0.528	0.415	0.476	0.448	0.376	0.425
695.8	25000000	1.462	0.629	0.541	0.658	0.494	0.511	0.596

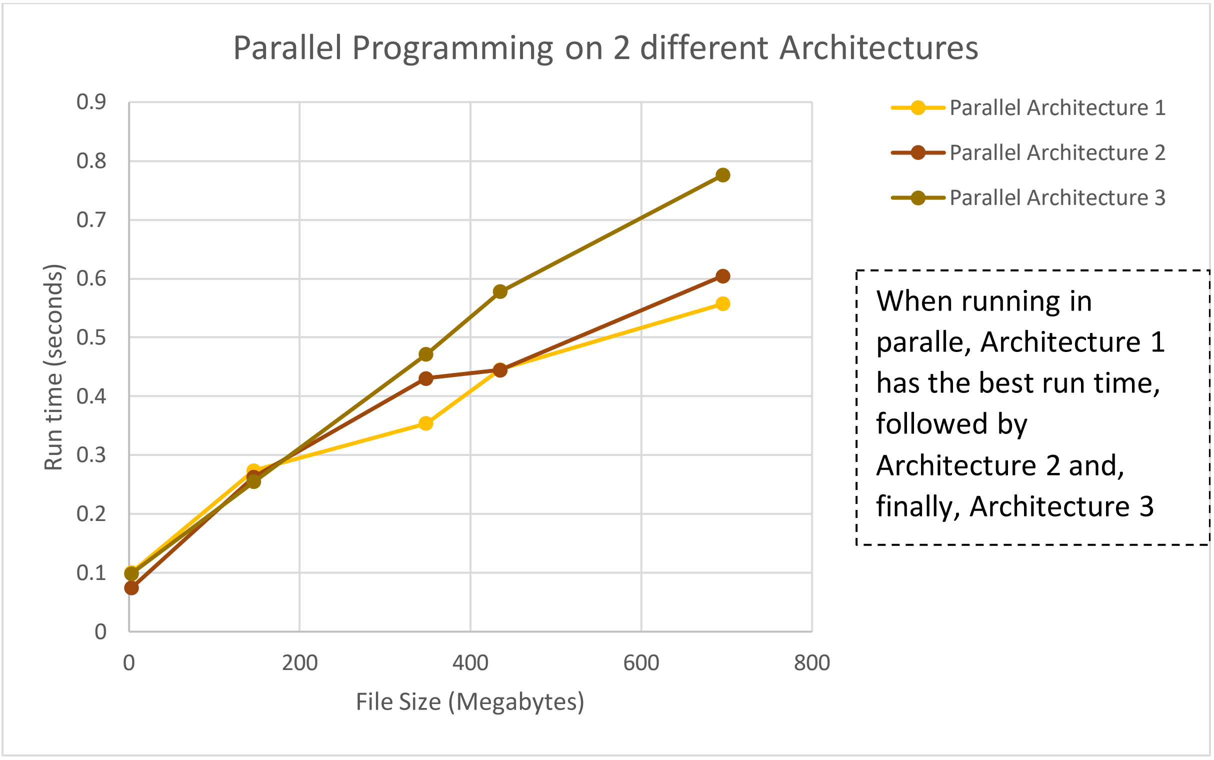
Architecture 3 : HP ProBook 450 G3 Notebook Intel® Core™ i5-6200 CPU @ 2.30GHz 2.40GHz

Input File		Sequential (seconds)	Parallel (seconds)					
			Sequential Cut-off					
Size(MB)	Dim()		100	500	1000	5000	10000	50000
2.784	100000	0.031	0.156	0.094	0.094	0.078	0.093	0.078
146.4	5242880	0.469	0.25	0.234	0.234	0.266	0.234	0.312
347.9	12500000	1.281	0.578	0.469	0.437	0.422	0.422	0.500
434.9	15625000	1.374	0.656	0.578	0.546	0.594	0.546	0.547
695.8	25000000	2.265	0.765	0.766	0.750	0.797	0.766	0.813

Based on the above tables, the following graphs of Run-time (y-axis) against Input File Size (x-axis) can be generated to visually demonstrate significant speed up brought through parallel programming.

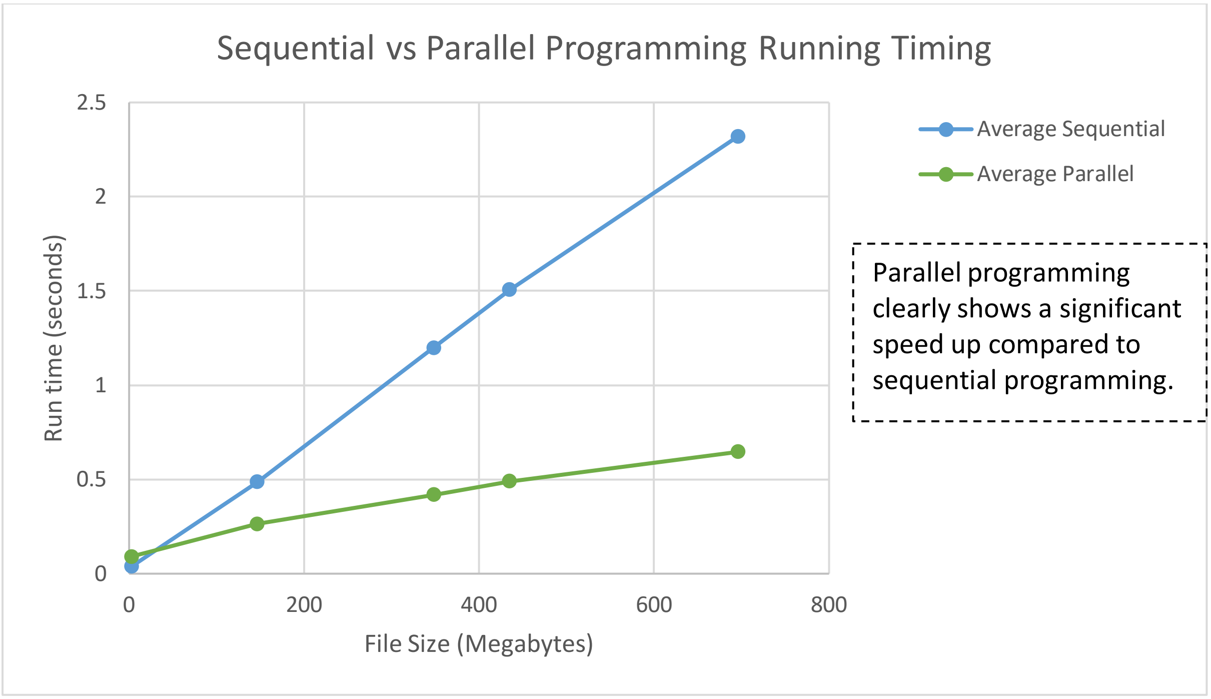


When running sequentially, Architecture 2 has the best run time, followed by Architecture 3 and, finally, Architecture 1



When running in paralle, Architecture 1 has the best run time, followed by Architecture 2 and, finally, Architecture 3

The 2 graphs above run using the same programming style and focuses mainly on comparing the run times by the different distinct machine architectures. For the aim of this project, the following graph will assess the difference between the architectures' average run time while using sequential programming and parallel programming.



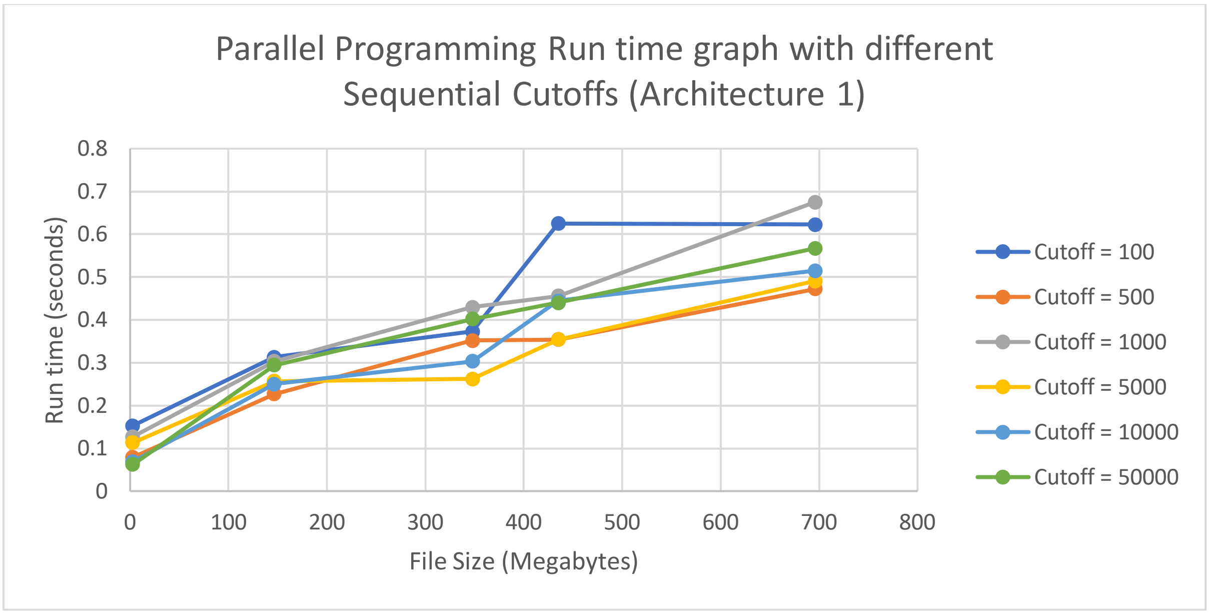
Discussion

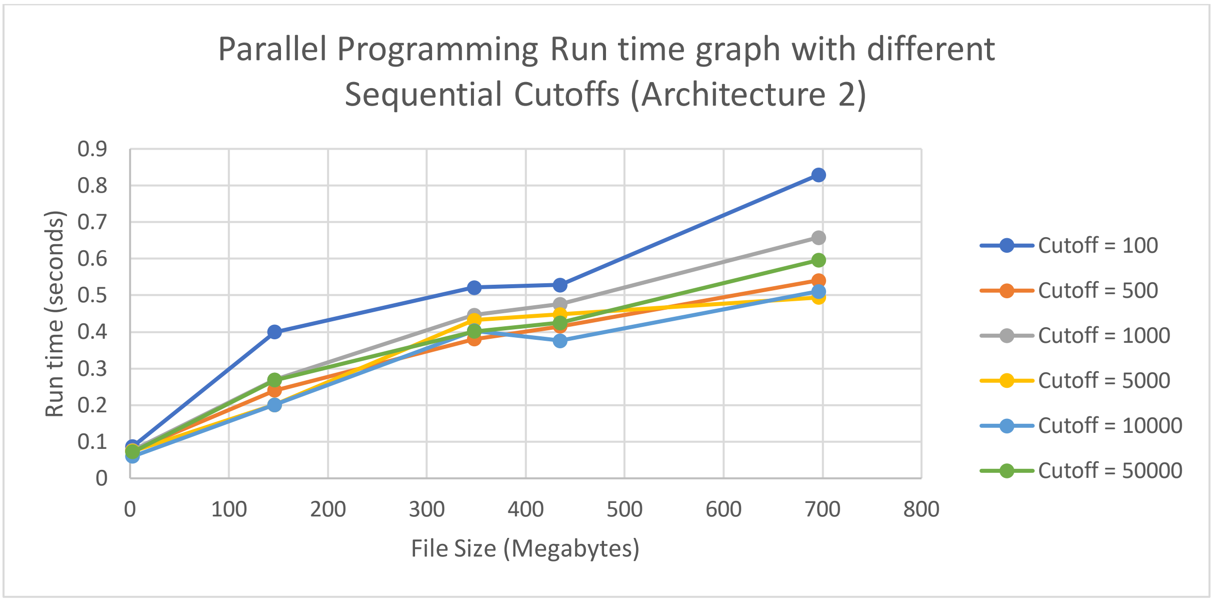
For the aim of this weather simulation project, parallelization is very worth the effort and computation as the input file containing multiple weather information per unit time is very large in size. Sequential programming, in this case, would consume too much time and hence, render weather data from the input files worthless. Therefore, tackling this issue with parallel programming seems to be the best approach towards data accuracy and algorithm performance.

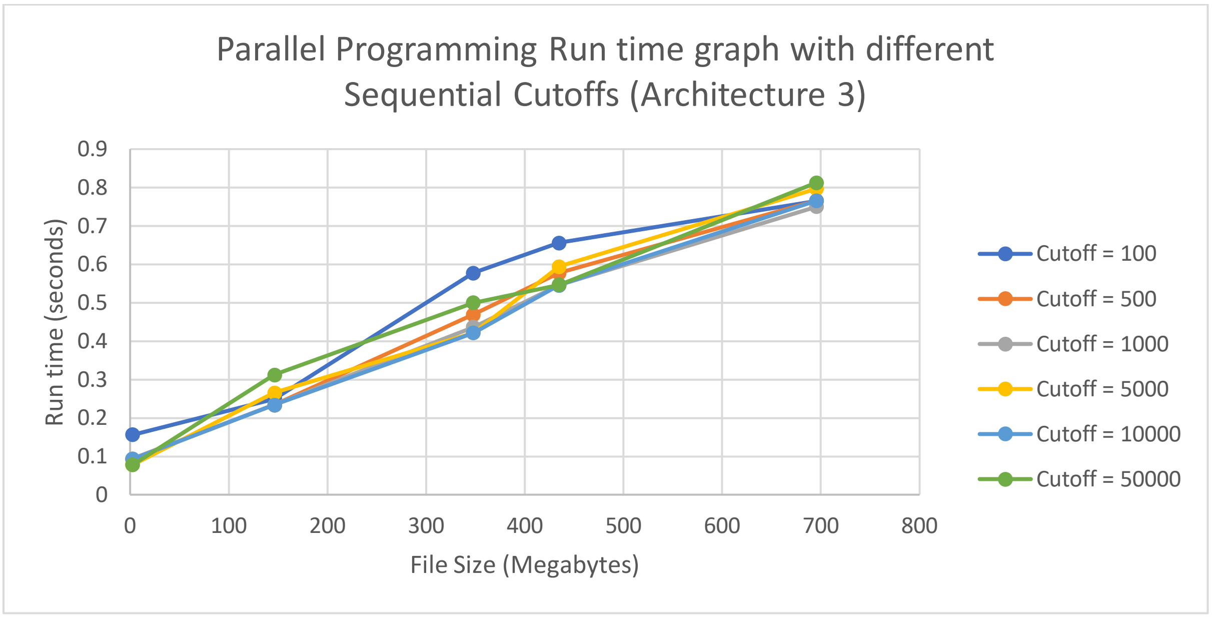
As per the intersection of the graphs above, parallel programming will provide a speed up compared to sequential programming as soon as the input file size exceeds 3.2 Megabytes. As the input file size increases, parallel programming proves to be more beneficial in terms of speed and performance. For example, according to my largest input file having a digital size of approximately 700 Megabytes, parallel programming provides about 3.75 times speed up as compared to the traditional sequential program. Theoretically, ideal parallel programming is supposed to bring forward the power of all the cores available. Thus, operating with a 4 cores machine should bring a speed up of 4 compared to sequential programming.

The sequential cut-off assigned in the parallel algorithm also has a major effect on the speed up depending on the input file size and the machine architecture used. The following page consists of 3 graphs constructed by the 3 different machine architecture used as the sequential cut-off is varied in the project.

MOHEE Bhavish
MHXBHA001







From the graphs above, the optimal sequential cut-off for each architecture could be deduced by the line which is relatively lower than the rest. The points connected by this line are, on average, lower than the run times with other sequential cut-offs. Therefore, the optimal sequential cut-off for the following architectures is:

- Architecture 1: Cut-off = 5000 (yellow line)
- Architecture 2: Cut-off = 10000 (pale blue line)
- Architecture 3: Cut-off = 1000 (grey line)

Now, to find the optimal number of threads on each architecture, a simple formula can be applied:

Optimal Number of Threads = Dim()/(Sequential Cut-Off – 1)

Hence,

- Architecture 1: Dim()/4999
- Architecture 2: Dim()/9999
- Architecture 3: Dim()/999

Conclusions

Parallel programming can have a huge positive impact on the speed performance of an algorithm. However, this programming style only exposes its benefits over a range of data set size. Within this range, the run time can become 3 to 4 times smaller than through the use of sequential programming. The speed up provided by parallel programming is dependent on the input size as well as the pre-set sequential cut-off.

According to my results, based on the 3 machine architectures this project was tested on, it can be deduced that each architecture is different in terms of number of cores or clock speed, which can affect speed up through parallel programming. The results presented in this report are quite reliable since the experiment has been conducted on 3 different architectures and agreed on the speed up that parallel programming provided.