**NGRBLE001 BLESSING NGORIMA**

**ASSIGNMENT 3**

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

## INTRODUCTION

An Introduction, comprising a short description of the aim of the project, the parallel algorithms and their expected speedup/performance

The aim of the project or program is to perform weather simulation for the purposes of real time prediction. It is vital to stress that weather prediction is a complex problem that need to be completed within a specific time limit in order to be useful. Parallel programming achieves this through its splendid rate of acceleration computer processes.

The prime focus of the program is comparing the serial method of calculating the prevailing wind direction and he types of clouds that might result with the parallel version. The basic task is to code a serial version and prevailing wind and cloud classification in java language.

The following is the requirements that needs to be archived to produce a working parallel version of the same serial or normal program:

- Code a parallel version of prevailing wind and class classification
- Use the Java Fork/Join framework
- Parallelize with a divide-and-conquer algorithm
- Benchmark using a high-precision timer by varying
  - Problem size
  - Machine
  - Number of threads or Sequential cutoff
- Using high-precision timing to evaluate the run times of your serial and your parallel method, across a range of input sizes, and Experimenting with different parameters to establish the limits at which sequential processing should begin.

It is important to time the execution of the parallel sorts across a range of data sizes and number of threads (sequential cutoffs) and report the speedup relative to a serial implementation.

Note that parallel programs need to be faster than the serial versions.

## METHODS

This section details the approach taken to successfully accomplish the given problem with the best results ever. The following are the algorithms used to make the predictions of the task at hand.

`public class CloudData {}`

CloudData contains attributes which are used by all the classes in the project. This class is useful for both the serial version and the parallel version of the program. CloudData instantiates an in plane regular grid of wind vectors that identifies with the name advection, a vertical air movement strength that identifies with the name convection that evolve over time and data grid dimensions. Each data element is a single layer of air at a particular time represented as a regular two-dimensional matrix with each matrix entry consisting of three floating point values. Advection is defined by The first two elements representing the x- and y-coordinate components of a wind vector (w).

This class is used to write classification output to file. Notice functions like `locate(int pos, int [] ind) and dim()` that give the exact position of a data element.

`public class Sequential Wind{}`

`Sequential_Wind` is the second class of the program and as the name clearly suggests it is the class implementing the serial version of the weather simulation problem. The class is a non-parallel stream that use just one thread to process the pipeline.

Firtsly Sequential_Wind serially calculates the average wind vector for all air elements by manipulating `cl.locate(k, ind_s)` method of CloudData in a single for loop. It is at this point where timing of the serial version of the simulation starts.

Secondly Sequential_Wind assigns to each air layer element an integer code (0, 1 or 2) that indicates the type of cloud that is likely to form in that location based on a serial comparison of the local average wind direction and uplift value. It uses five nested for loops sequentially to chief the second task altogether, and timing of the algorithm ends after the classification of clouds into appropriate codes.

`public class Wind extends RecursiveTask<Double>{}`

Wind class marks the beginning of the parallel version of the weather simulation problem. It extends `RecursiveTask<Double>`, a class used to create a Double return type. Wind gives the context of parallel programming as it encapsulates tasks that divide into independent subtasks and the final outcome of the task can be obtained from the outcomes of the subtask.

This class contains `protected Double compute()` which has the code that represents the computational portion of the task and overrides the **compute()** of RecursiveTask.

Wind class calculates the average wind vector for all air elements and assigns to each air layer element an integer code (0, 1 or 2) that indicates the type of cloud that is likely to form in that location based on a comparison of the local average wind direction and uplift value, but all the is done using recursion in the java language.

```
public class WindAvg{}
```

Inside the WindAvg class there is an importation of ForkJoinPool that contains classes like ForkJoinTask, a base class of RecursiveTask. A ForkJoinPool is constructed with a given target parallelism level; by default, equal to the number of available processors. The pool attempts to maintain enough active (or available) threads by dynamically adding, suspending, or resuming internal worker threads, even if some tasks are stalled waiting to join others. This class shows the time taken to complete the task for different input sizes.

**This section must explain how you validated your algorithm (showed that it was correct), as well as how you timed your algorithms with different input, how you measured speedup:**

The first step for validating the algorithm is to compare the output produced after providing simplesample_input.txt file. Make sure that the average wind vector matches the one provided on the problem for both serial and parallel versions of the algorithm.

Performance measure is implemented by looking at the difference in run times for the serial and parallel method, across a range of input sizes supplied. Another experimenting technique used for this specific problem is to record time versus input size for six different threads or sequential cut-off values.

Looking at the parallel method within WindAvg.java in particular the speedup is measured from the moment forkJoinPool invokes Wind.java which performs the parallel computation till it ends.

Timing must is restricted to the cloud classification and prevailing wind code and excludes the time to read in the files and other unnecessary operations.

Note that parallel programs are faster than the serial versions.

**The machine architectures you tested the code on and interesting problems/difficulties you encountered:**

**The architectures used:**

1. **Acer Spin 5 laptop: intel core i7, 7$^{th}$ Generation, 6 cores**
2. **Uct CSC Senior Lab Desktop : intel core i5, 8$^{th}$ Generation, 4 core**

**Problems encountered:**

- Notice that the speed up for serial or time of execution for both method appear to change rapidly for the same input data for some time.
- Encountered a problem with other applications on my personal computer when I was running the parallelized algorithm. Microsoft office word crashed and forced to shut down resulting in the loss of my initial report.

- The speed up time or time of execution printed out by the program does not match the time I waited to write out put on the screen or file for both architectures.
- Other processes on the Lab computer performed normally compared to processes on my laptop as some applications would take long to open.
- IAM CONFUSED WHY IS I5 PERFORMING BETTER THAN I7

## RESULTS AND DISCUSSION

The following are the raw results demonstrating the effect of data sizes and numbers of threads and different architectures on parallel speedup. This section should include speedup graphs and a discussion.

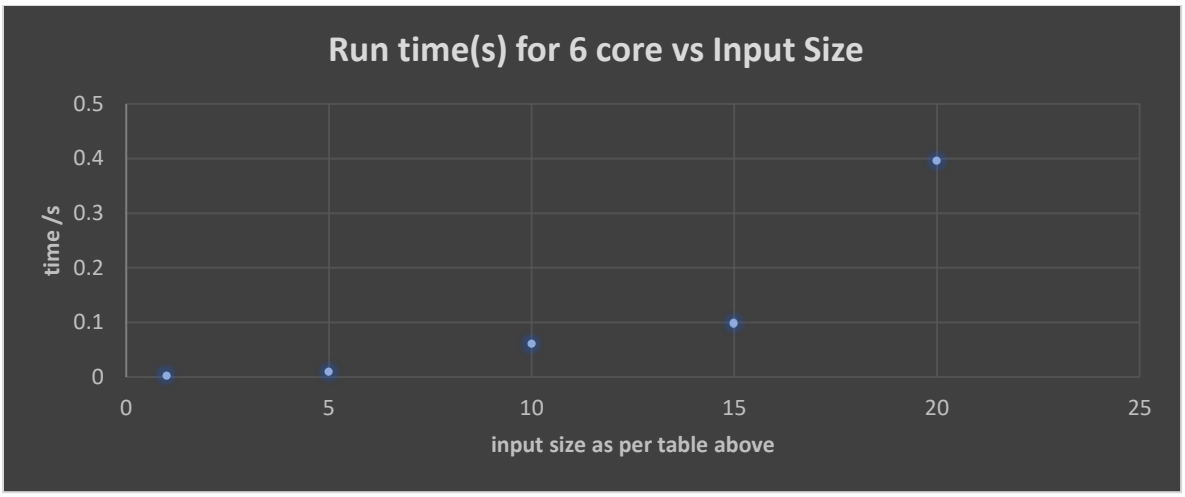
### Sequential Program

A range of input sizes is supplied to the sequential program to establish high-precision timing for evaluating the run times. Experimenting with different parameters to establish the limits at which sequential processing should begin.

***Table1. : Shows the range of input file sizes with respective run times for sequential method. This is for Architecture 1 (Acer Spin 5 laptop: intel core i7, 7$^{th}$ Generation, 6 cores ) and Archicture 2 ()***

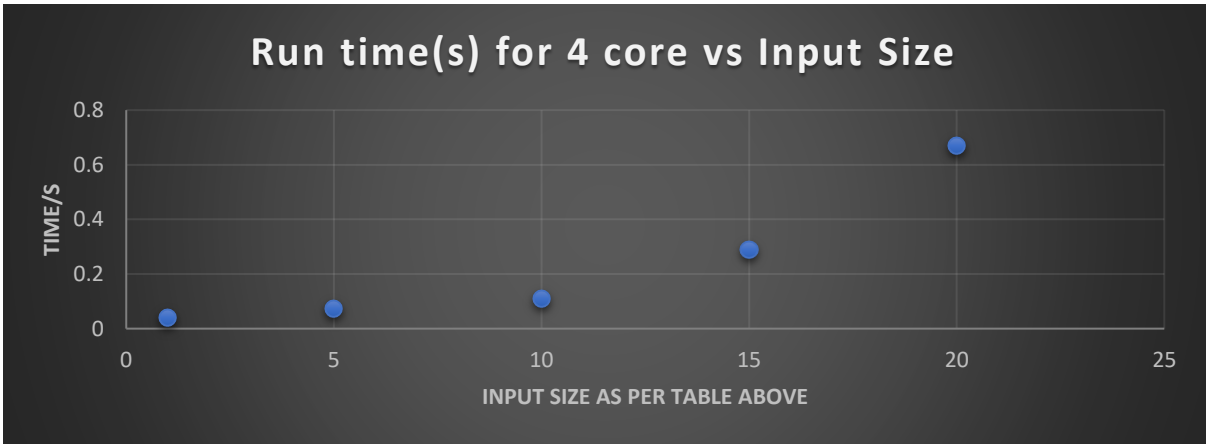
| **Input Size** | 1*32*32 | 5*64*64 | 10*128*128 | 15*256*256 | 20*512*512 |
|---|---|---|---|---|---|
| **Run time(s) For 6 core Architecture1** | 0.003 | 0.009 | 0.06 | 0.098 | 0.395 |
| **Run time(s) For 4 core Architecture2** | 0.04 | 0.071 | 0.109 | 0.290 | 0.670 |

**Graph1 : Run time(s) for 6 core vs Input Size**



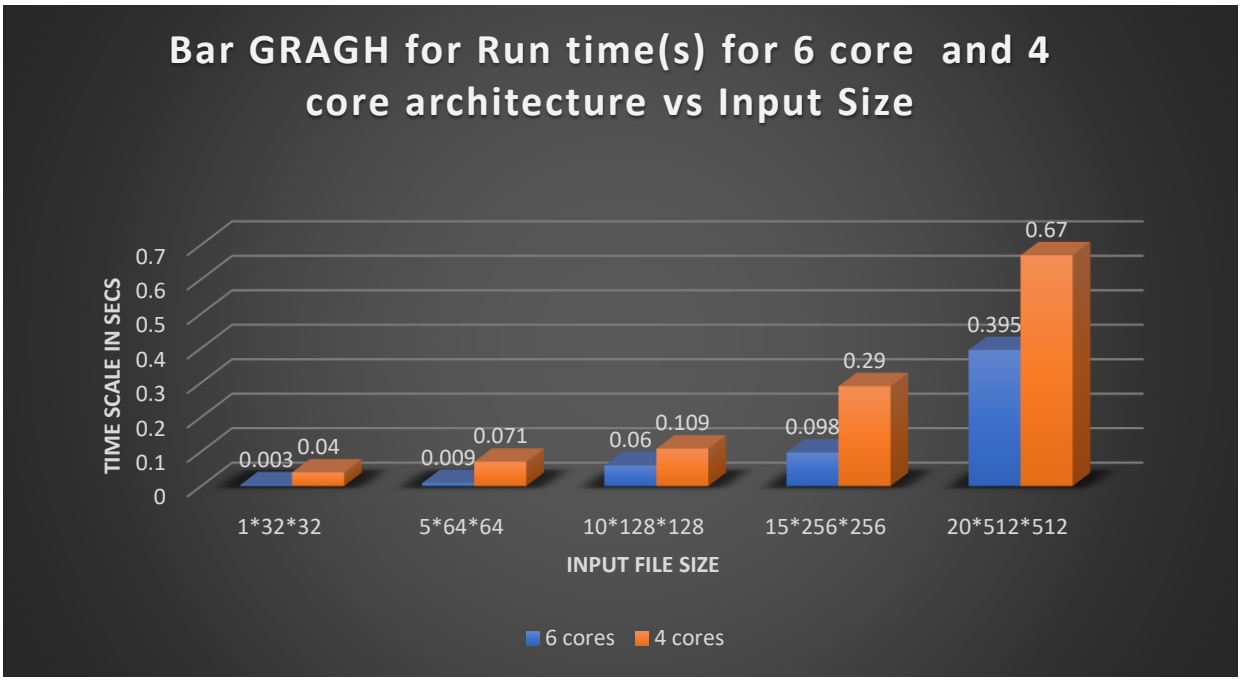

The run time starts to rapidly increase after the input size of 15*256*256 and the algorithm is not that bad in operation. This is an exponential graph of run time against different input file sizes.

**Graph 2: Run time(s) for 6 core vs Input Size**




The run time starts to rapidly increase after the input size of 10*128*128and the algorithm is slower copered to 6 core architecture. This is an exponential graph of run time against different input file sizes.

**Graph 3: Bar GRAGH for Run time(s) for 6 core and 4 core architecture categorized in Input Size**




*The graph clearly shows that the architecture with more takes less time to compute the same task. This* displays the power of processors and architecture bench mark.

This graph shows the best speed up between the two architectures.

**Parallel Program**

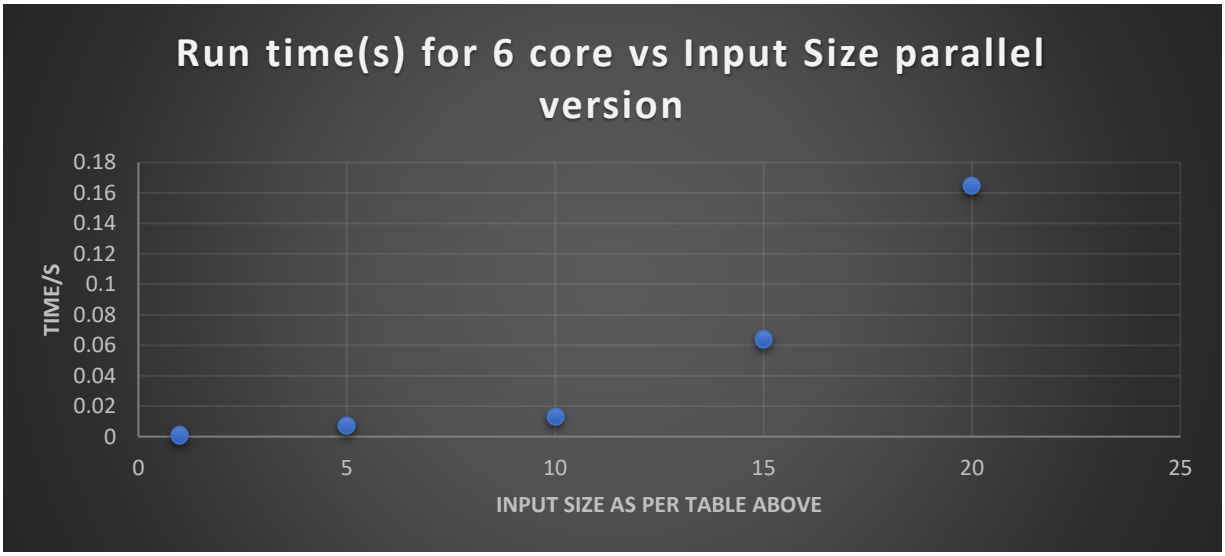
A range of input sizes and various number of threads (sequential cut-offs) is supplied to the parallel program to establish high-precision timing for evaluating the run times. Experimenting with different parameters to establish the limits at which parallel processing should begin.

**Table 2:** ***Shows the range of input file sizes with respective run times for sequential method. This is for Both Architecture 1*** (Acer Spin 5 laptop: intel core i7, 7th Generation, 6 cores ) ***and Archicture 2*** *(* Uct CSC Senior Lab Desktop : intel core i5, 8th Generation, 4 core *)*

The sequential cut-off is contant here!!

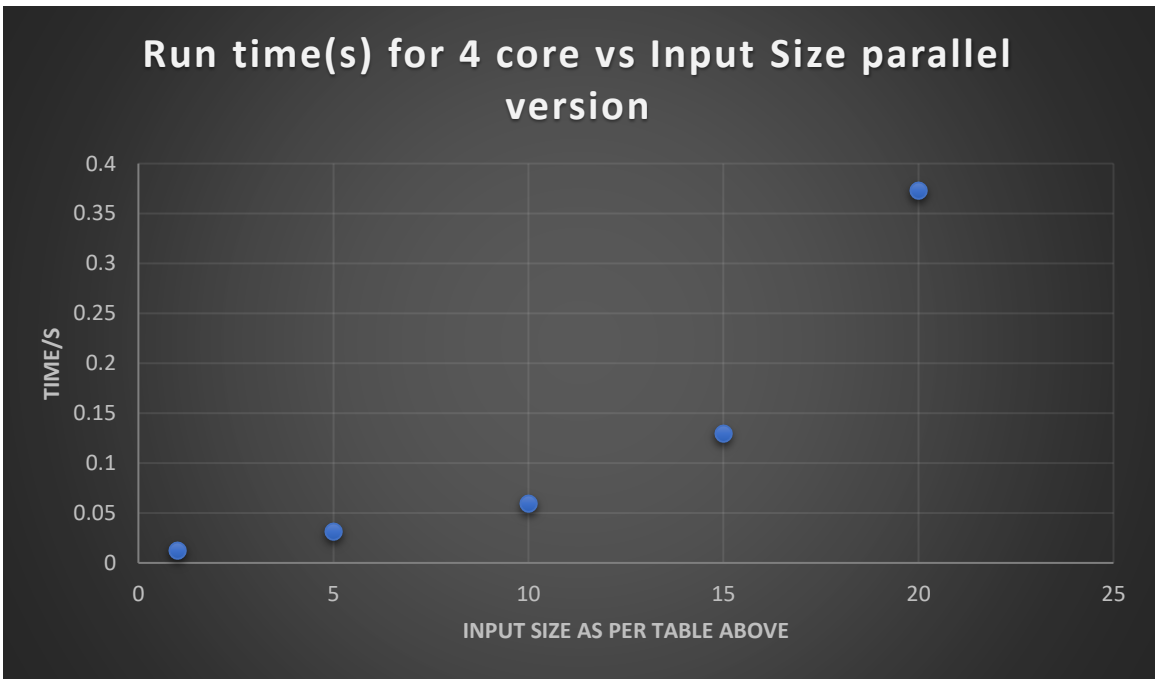
| **Input Size** | 1*32*32 | 5*64*64 | 10*128*128 | 15*256*256 | 20*512*512 |
|---|---|---|---|---|---|
| **Run time(s) For 6 core Architecture1** | 0.001 | 0.007 | 0.013 | 0.064 | 0.165 |
| **Run time(s) For 4 core Architecture2** | 0.012 | 0.031 | 0.59 | 0.129 | 0.373 |

**Graph 4: Run time(s) for 6 core vs Input Size parallel version**



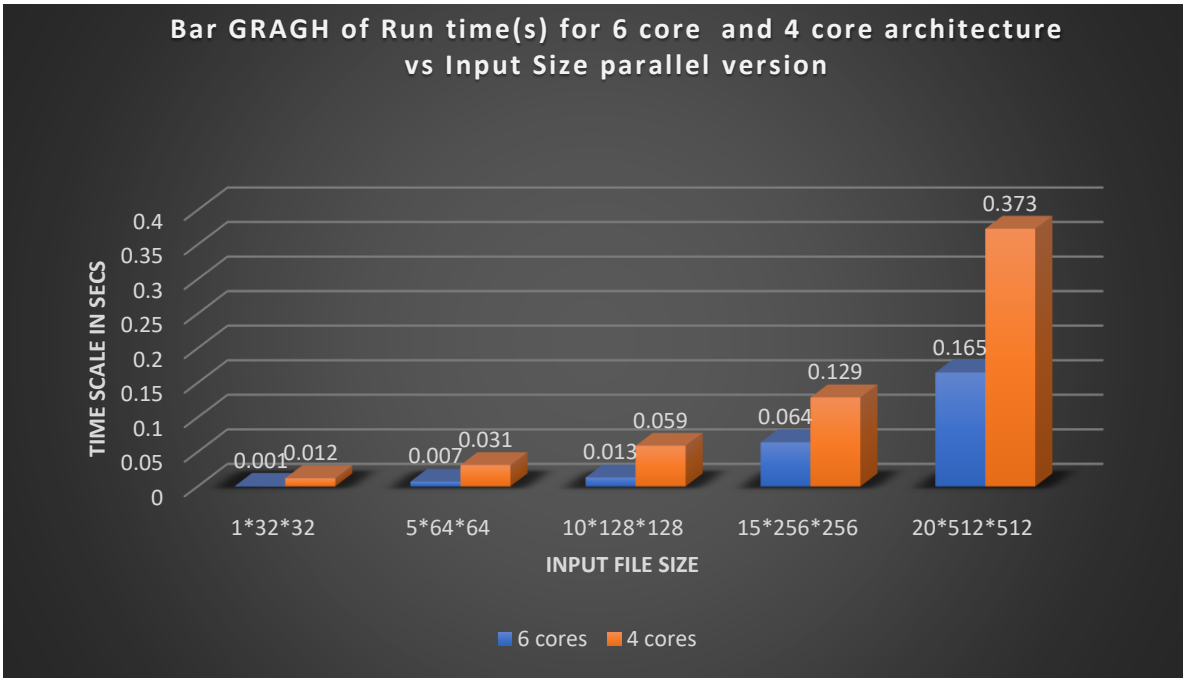

Shows an exponential relationship between time and input size using a constant sequential cut-off.

**Graph 5: Run time(s) for 4 core vs Input Size parallel version**




Shows an exponential relationship between time and input size as the run time rapidly increases at the 15$^{th}$ marking of the x-axis.

**Graph 6 : Bar GRAGH of Run time(s) for 6 core and 4 core architecture vs Input Size parallel version**



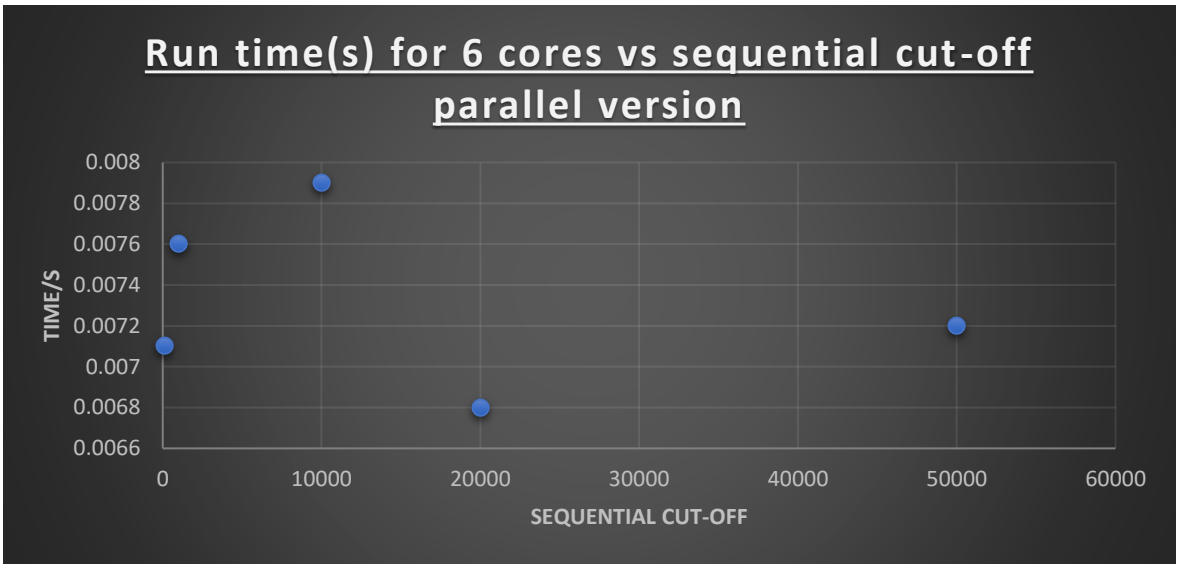

This graph shows the difference in performance for the two different processors at once.

**Table 3:** ***The table shows values of 5 different sequential cut-off data in relation to input sizes size of 1*30*30 and the run time. This is for Architecture 1*** (Acer Spin 5 laptop: intel core i7, 7th Generation, 6 cores ) ***and Archicture 2*** *(* Uct CSC Senior Lab Desktop : intel core i5, 8th Generation, 4 core *)*

The sequential cut-off is contant here!!

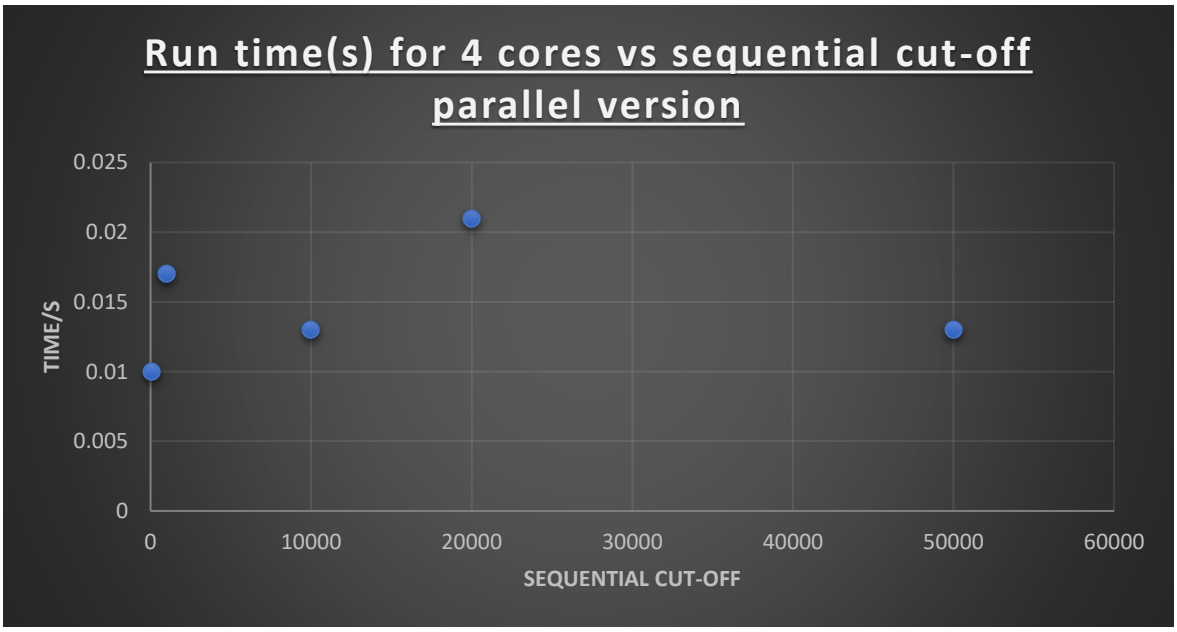
| **Sequential cut-off** | 100 | 1000 | 1000 | 20 000 | 50 000 |
|---|---|---|---|---|---|
| **Run time(s) For 6 core Architecture1** | 0.0071 | 0.0076 | 0.0079 | 0.0068 | 0.0072 |
| **Run time(s) For 4 core Architecture2** | 0.010 | 0.017 | 0.013 | 0.021 | 0.013 |

**Graph 7: Run time(s) for 6 cores vs sequential cut-off parallel version**



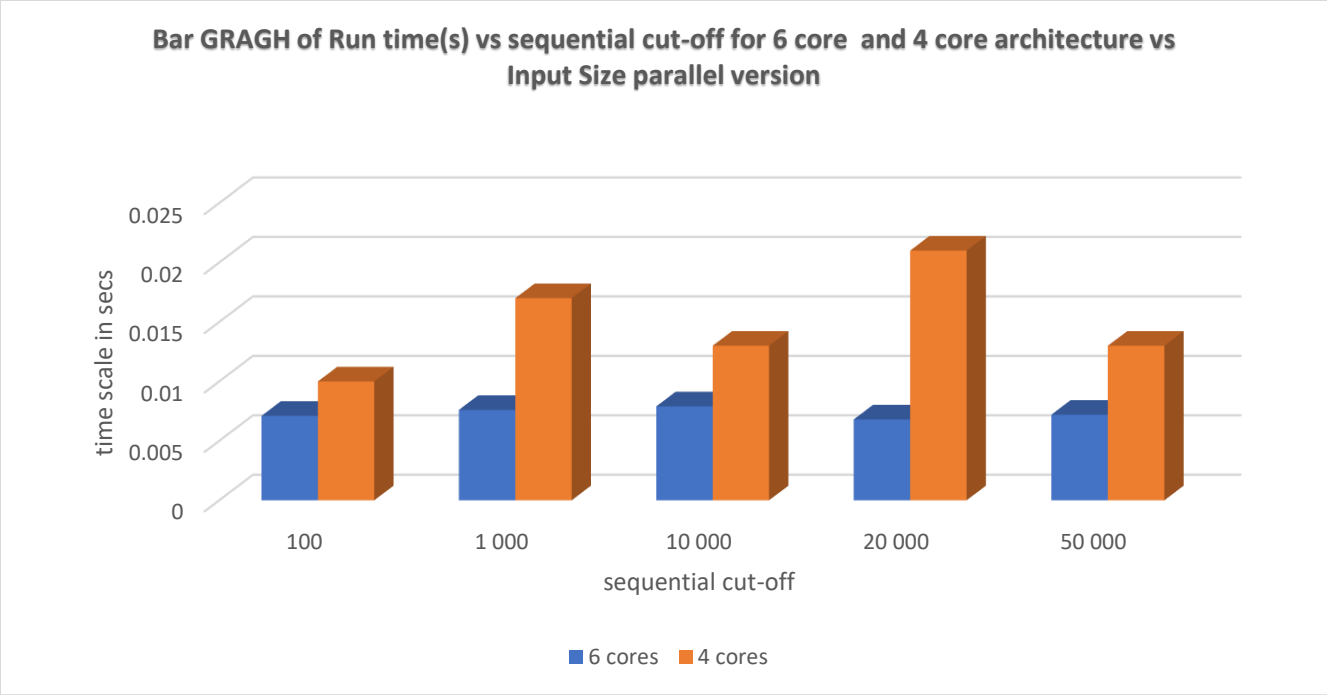

This is a harp hazard distribution produced by different values of cut- off.

**Graph 8: Run time(s) for 4 cores vs sequential cut-off parallel version**




This graph shows a Harp hazard representation of the combination of run time and sequential cut-off values.

**Graph 9: Bar GRAGH of Run time(s) vs sequential cut-off for 6 core and 4 core architecture vs Input Size parallel version**




Shows a normal distribution of run time values chosen for a specific input size over a range of sequential cut-off values.

**Discussion**

**It is clearly worth it to tackle this problem by parallelization(multithreading) as the tasks is divided and processed way faster than serially.**

**The parallel program best performs best at wide range of input sizes, that is :**

| | | |
|---|---|---|
| 10*128*128 | 15*256*256 | 20*512*512 |

**The optimal sequential cut-off for this problem is displayed as 20 000 according to the results. It is important to notice from graph 9 that each architecture has a different optimal number of threads. It is believed that for each core there is a thread so the number of threads for the 4 core architecture is 4 and for the 6 core architecture is 6.**

## CONCLUSIONS

**A Conclusions (note the plural) section listing the conclusions that you have drawn from this project. What do your results tell you and how significant or reliable are they?**

Presented with the facts of the experiment and outputs produced whilst testing, one can recommend that the task of weather simulation for the purpose of prediction should be tackled using parallelization

or multithreading. This claim is supported by the experimental data displayed in the results and Discussion section.

Firstly, consider benchmarking using a high- precision timer by varying problem size and machine architecture for both the serial and parallel methods:

- In overall the architecture with more cores (6 core machine) produced better speed-up as compared to the 4 core machine.
- The 6 core architecture managed to read in data from large input file sizes and write it out as will.
- The sequential method approximately similar to the serial version for simple or small input text files.

Secondly, consider the comparison of the sequential version of the program and the parallel version:

- The parallel version has good run times across a wide range of input sizes mainly because the algorithm performs a divide-and-conquer approach to the tasks.
- The two methods perform almost similar up until the sequential cut-off becomes 100