CSC2002S Assignment 3

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

Samantha Ball
BLLSAM009

3 September 2019

Introduction

Aim

The aim of this assignment is to investigate the benefits of parallel programming by performing weather simulation analysis on a large data set. Parallel processing is most beneficial when processing particularly large amounts of data and thus the extensive data used in weather prediction render it a suitable task for the analysis of parallel programming efficiency. The experiment will be run over varying data sizes, machine architectures and values of sequential cut-off in order to determine the optimal parameters for parallel computation in this context.

Computational Problem

The parallel algorithm will be required to compute the prevailing wind direction over the entire dataset as well as the cloud type most likely to form at any given data point. The input data will be given as a set of 3-dimensional arrays containing information regarding the co-ordinates of the wind vector, known as advection, and the rate at which air shifts upwards or downwards, known as convection. The data will also be given over several time steps. The expected output of the program comprises the x and y co-ordinates of the prevailing wind vector averaged over all simulation data and a 3-dimensional array containing integers representing the cloud classification at each grid-points. Clouds will be classified according to the comparison of the wind magnitude to the uplift value, and may be given the value 0, 1 or 2, representing cumulus, striated stratus and amorphous stratus cloud types respectively.

Parallel Algorithm Structure

A well-known parallel algorithm structure is the divide-and-conquer algorithm in which the data set to be processed is first divided between multiple threads and then the results of the operation performed by the threads are combined until the final answer is reached. The divide-and-conquer algorithm thus relies on a multi-branched recursion structure where the problem is broken into smaller sub-problems which are each given to a new/separate thread. This structure can be used for the above stated computational problem in order to perform the addition of all wind vectors, used in finding the prevailing average wind, as well as the writing of cloud classification data to the 3-dimensional classification array.

The Java Fork/Join framework will be used as a lightweight alternative to the Java Thread class. This is due to the large number of threads needed in a divide and conquer algorithm and thus the need to minimize the overhead required to create each thread. This causes the Fork/Join framework to be particularly advantageous for the weather simulation analysis due to its lightweight structure and decreased overhead.

Expected Speed-Up

Through the use of the divide-and-conquer strategy, the projected speed up of the parallel program with respect to the original sequential version can be estimated using time complexity. This expected speed up is calculated by comparing the O(n) performance of the sequential algorithm to the theoretical O(n/P + log n) performance of the parallel algorithm, where n represents the number of input grid elements and P represents the number of available processors.

$$Theoretical\ speed\ up = \frac{O(n)}{O(\frac{n}{P} + \log(n))}$$

Or for an infinite number of processors

$$Theoretical\ speed\ up = \frac{O(n)}{O(\log(n))}$$

It should be noted that speed up is highly dependent on the number of processors available and thus the properties of the machine on which the analysis is run will play a large role in the results obtained. For a sufficient number of processors, the divide-and-conquer parallel algorithm is expected to run with a time complexity of O(logn), which provides exponential speed-up over the original sequential time complexity of O(n), where n is the total number of grid-points given as an input. However in practice this may be limited by the number of processors available, as well as the particular sequential cut-off value chosen.

Methodology

In order to solve the computational task, a sequential version of the algorithm was first created as a benchmark against which to compare the speed up of the parallel algorithm. The sequential algorithm was comprised of two core methods, *findAverage* and *getClouds,* which calculate prevailing wind direction and cloud type respectively. The input data used in the prevailing wind and cloud classification algorithms is first read in from a given input file containing data detailing the advection and convection in a single air layer over time.

In order to find the prevailing wind over the entire data set, the following algorithm structure was used:

```
Vector findAverage()
{

    initialize wind vector;
    initialize xsum, ysum, numPoints = 0;

    for(each time step in the advection array)
      for(each x co-ordinate in the advection array)
         for(each y co-ordinate in the advection array)
         {
            add corresponding x-coordinate to xsum
            add corresponding y-coordinate to ysum
            increment numPoints to keep count of data points
         }
    xAverage = xsum/numPoints;
    yAverage = ysum/numPoints;
    add xAverage, yAverage to wind vector;
```

```
    return wind vector;
}
```

It can be observed that the algorithm uses a single thread to iterate through each item in the grid and add the x and y components of the advection array to the x and y sums respectively.

Similarly, in order to find the cloud type for each grid point and write it to the classification array, the following method structure was implemented:

```
void findCloud()
{
    initialize xsum, ysum, numPoints = 0;

    //FIND local average for element at index (i, j)
    for(each x co-ordinate from i-1 to i+1)
      for(each y co-ordinate from j-1 to j+1)
        {
           check if element is within bounds;
           if (inbounds)
           {
              add corresponding x-coordinate to xsum
              add corresponding y-coordinate to ysum
              increment numPoints to keep count of data points
           }
        }
    xAverage = xsum/numPoints;
    yAverage = ysum/numPoints;
    compute magnitude of local wind average from xAverage and yAverage;
    find uplift value at specified co-ordinates;

    //CLASSIFY cloud types based on comparison
    if (wind magnitude < uplift value)
           cloudType = 0;
    else if ((wind magnitude > 0.2) AND (wind magnitude > uplift value))
           cloudType = 1;
    else
           cloudType = 2;

    write cloudType to classification array;
}
```

The above algorithm is then run for every grid-point in the advection array, using a triple for-loop structure as in case of the *findAverage* method. Hence once again, a single thread is used to iterate through each item in the grid and thus only one grid-point can be operated on at a time. It should be noted that the *findCloud* method calculates the local average wind for use in the cloud classification comparison. This is implemented by finding the average of the eight neighbouring data points centered on the grid-point in question. This is significant because only nine data points are involved in this calculation and thus the summation required will not be parallelized in the later parallel algorithm.

The results of both the *findAverage* and *findCloud* methods are then written to an output file.

Parallelization

In the case of the parallel algorithm, the *findAverage* and *getClouds* method equivalents were initially created as two separate thread classes, one of type RecursiveTask and one of type RecursiveAction. The *findAverage* equivalent was created as a RecursiveTask as it returns a vector containing two double

values. The *getClouds* equivalent was of type RecursiveAction as it did not return a value but instead modified the contents of the classification array. However, in order to optimize for maximum efficiency, the two thread classes were combined into one thread class called *WriteClouds. (*In this way, the already created threads were used for all necessary operations instead of requiring the additional overhead of creating new threads for a separate task.) Since *WriteClouds* combines the functionality for both methods, it is of type RecursiveTask as it returns a vector of doubles used in finding the wind average. The overall algorithm thus combines both the reduction and map algorithm patterns as it produces a single answer from a collection and operates on each element of a collection independently to create a new collection of the same size. The outline of the *WriteClouds* compute method is shown below:

```
Vector compute()
{
    if ((hi-lo) < SEQUENTIAL_CUTOFF)
     {
        initialize new gridpoint;
        initialize vector to contain xsum and ysum;
        for(each index from lo to hi-1)
        {
           convert linear index to grid-point;
           find cloud type for corresponding grid-point;
           add corresponding x-coordinate to xsum;
           add corresponding y-coordinate to ysum;
        }
        add xsum, ysum to sum vector;
        return sum vector;
    }
  else
    {
         left = new WriteClouds object initialized with first half of array
         right = new WriteClouds object initialized with last half of array
         left.fork(); //create new thread
         rightAns = right.compute(); //use current thread to run compute
         leftAns = left.join(); //wait for other thread to complete
         initialize answer vector;
         add (leftAns + rightAns) to answer vector;
         return answer vector;
     }
}
```

In the above pseudocode, the application of the Fork/Join framework can be observed. During the divide stage, since the number of elements to be processed by a particular thread is greater than the sequential cutoff, new left threads are created using left.fork(). The current thread is then used to run the compute() method. This is significant in order to halve the number of threads needed by fully utilising the current thread. Hence only one new thread is created at each step. This process is repeated recursively until the sequential cut-off is reached at which point the array elements within that subsection are processed.

In order to process each array element, the given index in the overall linear array is converted to the indices of the 3-dimensional grid-point it represents. In this way the element can be mapped to its corresponding 3-dimensional representation in terms of time, x co-ordinates and y co-ordinates. As in the case of the sequential program, the cloud type for the given grid-point is determined and then written to the classification array. The central difference between the sequential and parallel versions is that separate threads are now performing the classification concurrently, instead of one thread iterating over all grid-points one at a time.

Similarly, within the same loop, the x and y co-ordinates of the given grid-points are added to their respective sums for use in the average wind calculation and a vector containing the x sum and y sum is then returned. Thus, the central advantage is the same as for cloud classification as the summing operation is performed by multiple concurrent threads instead of a single thread iterating through the entire 3-dimensional array. However, in this case the operation returns a value which must then be combined with the values returned by other threads in the combine stage. This is implemented through the use of left.join() which waits for the other thread to finish its execution and return the corresponding value. This value is then added to the value obtained by the current thread and the overall sum is returned.

A fundamental component of the parallel algorithm is the translation between the linear array of all data elements and the position in the 3D grid that a particular element represents. This allows only one high and low cut-off to be defined rather than a separate high and low cut-off for each dimension.

To start the creation and running of threads, a pool of threads known as the ForkJoinPool is instantiated and invoked. This structure allows the threads to have access to shared memory and enables efficient processing by managing worker threads and ensuring threads are not idle by taking new work from the ForkJoinPool.

Validation of Algorithm

In order to verify that the results of the above solution were correct, a program called *checkOutput* was initially written in order to read in the generated output file from the parallel algorithm and the expected output file and compare them. The files were compared on a byte by byte basis, checking equality in both the output data and length of the file. This program was also used to verify whether the sequential and parallel results corroborated one another. In this way, the correctness of the sequential and parallel algorithms was verified.

However, in order to check each iteration of the tests run for benchmarking, the file containing the known correct data was read in at the beginning of the program together with the input file. After each test iteration, the output data in each array was verified against the known correct data. In this way, each test iteration could be verified for accuracy of results.

Benchmarking

In order to determine the optimal parameters for the use of the parallel algorithm, several benchmarking tests were run. The time taken for finding the average wind vector and finding and writing the cloud types to the classification array was timed using the `System.currentTimeMillis` method. This method returns the current time in milliseconds and can thus be used to measure the time taken to

perform a particular operation by storing the time at which the operation was started and completed and then finding the difference between them. All timing measurements are thus given in milliseconds. The time taken to read and write data to and from the text files was explicitly excluded from the measured time in order to directly compare only the time taken to process the data.

The variables investigated included the size of the input data file, the number of threads as dictated by the sequential cutoff and the machine architecture on which the program is run. The program was tested with different input data sizes varying from 262144 grid elements to 5242880 grid elements, with the input size doubling for each iteration. This was achieved by reading in the whole data file once and then performing the desired operations on subsets of the data. Similarly, the sequential cutoff was varied from 50 to 500 000 in steps of unequal size, in order to determine the optimal balance between the number of threads and the resultant overhead. Lastly, the program was run on four different machine architectures, including

- a MacBook Pro 2017 with a dual-core 2.3GHz Intel Core i5 processor
- a Raspberry Pi 3B+ with a quad-core 1.4-GHz ARMv8 CPU
- a MacBook Pro 2015 with a dual-core 2.7GHz Intel Core i5 processor
- an Asus G73S with a quad-core 2GHz Intel Core i7 processor

Each variable was tested by keeping all other variable constant and then iterating through all test values, recording both correctness and speed. In order to attain results of greater accuracy, each iteration was run seven times and an average of the last five measured runtimes was calculated. The first two values were discarded to account for the effects of cache warming. The speed up due to the varying of each parameter as well as overall speed up against the sequential algorithm was then determined.

Challenges

A particular challenge encountered involved the limited memory available on the Raspberry Pi 3B+ as this caused a heap space error when trying to read in the input file. In order to overcome this and obtain comparable results, the input file was limited to a quarter of its full size. Comparable tests were then run on the MacBook Pro 2017 using only a quarter of the file size in order to accurately compare performance.

Extension

The investigation was extended to more than two architectures, including a Raspberry Pi 3B+, in order to more thoroughly investigate the tradeoffs between memory limitations, clock frequency and available cores. As an additional investigation, the effects of calculating the local average in the *findCloud* method over a larger sample of data was investigated. This was achieved by increasing the number of neighbouring data points included in the average from 8 neighbouring points to 24 and 48 neighboring points. The effects of this increase were then observed with respect to both accuracy of results and speed up.

Results and Discussion

Varying Data Sizes

By doubling the data size each iteration, the following results shown in Figure 1 were obtained. It can be observed that the parallel algorithm was more efficient than the sequential algorithm, as it achieved approximately 3x speed up at each respective data size.

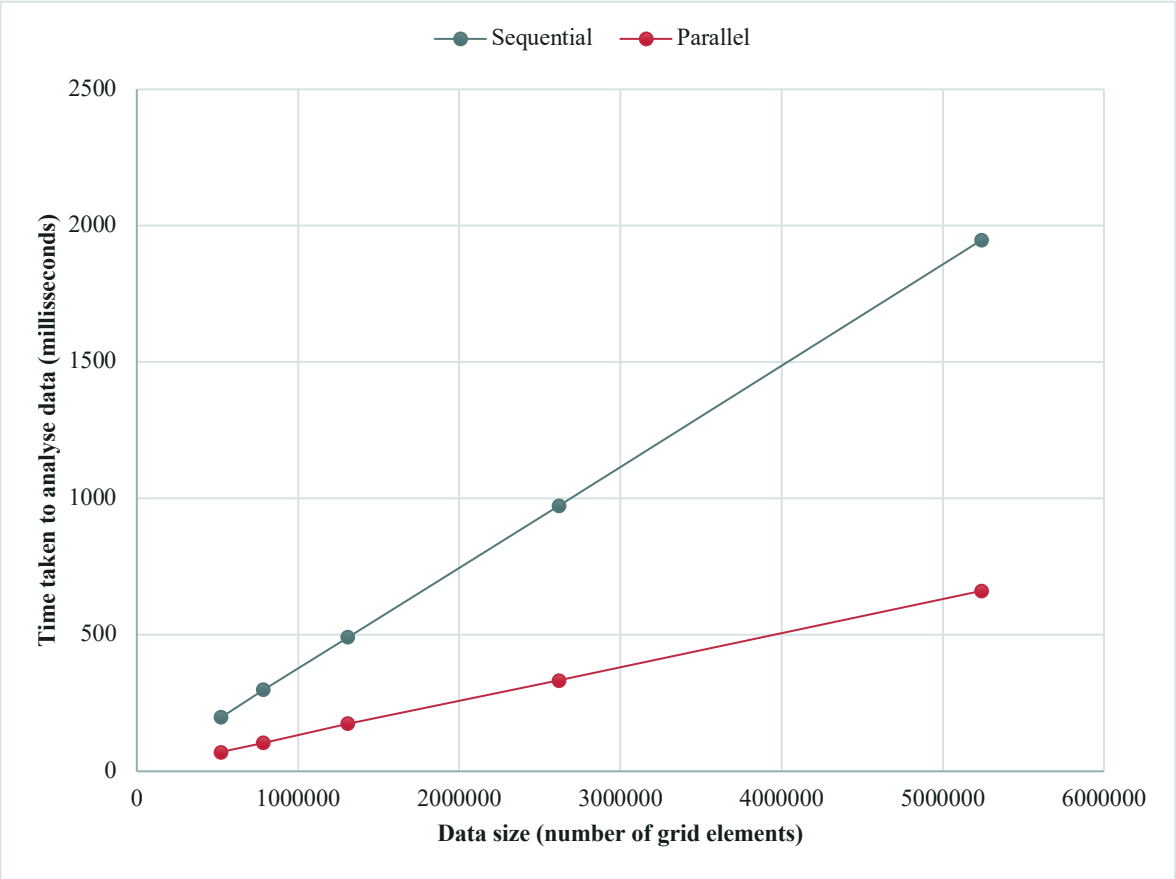

Figure 1: Graph showing run time of analysis with respect to increasing data size, for both sequential and parallel algorithms

This observed speed up value is verified by the tabulated averages below. As expected, for an increased data size, the run time of the analysis algorithm for both the sequential and parallel methods increases. However the rate at which the sequential program increases as a function of data size is much greater than for the parallel method, as can be seen by the slopes of the graph.

Datasize [no. of grid elements]	Run time (Sequential) [ms]	Run time (Parallel) [ms]	Speed up
262144	152	105	1,45
524288	198	71	2,78
786432	300	105	2,87
1310720	491	176	2,80
2621440	974	334	2,92
5242880	1948	662	2,94

The above results indicate that the speed up achieved by the parallel algorithm was just under 3x the sequential algorithm. Hence the average speed up over all data sizes was **2.86.** This value can be compared to the theoretical speed up obtained from the following equation,

$$Theoretical\ speed\ up = \frac{O(n)}{O(\frac{n}{P} + \log(n))}$$

Hence the expected speed up for the full input file size of n = 512*512*20 = 5242880 grid elements for a computer with 2 processors is

$$Theoretical\ speed\ up = \frac{5242880}{\frac{5242880}{2} + \log(5242880)} = 2$$

The measured speed up of 2.86 thus appears to exceed the theoretical speed up of 2. This may be due to inefficiencies in the sequential code that cause it to run slower than O(n) efficiency and thus distort the measured speed up.

In order to find the limits for which the parallel program performs better than the sequential version, increasingly small input data sizes were tested, with the smallest input data size being 262144 grid elements. It can be observed that as the data size decreases, the speed up realised by the parallel program decreases and thus there is a particular range over which the parallel algorithm is best suited. The limits of this range was found to be an input size of **262144 elements,** where the speed up drops significantly to 1.45. It should be noted that this limit was found for a fixed sequential cut-off value of 2500.

Investigating Sequential Cut-offs

By varying the value of the sequential cut-off from 50 to 50 000, the optimal value of sequential cut-off can be ascertained. The graph below indicates that a sequential cut off of **10 000** is most optimal as it results in the fastest run time. Hence, for the given dataset size and machine architecture, the optimal trade off between number of threads and thread overhead is given by a sequential cut-off of approximately 10 000. This cut-off value corresponds to approximately **7159** threads.

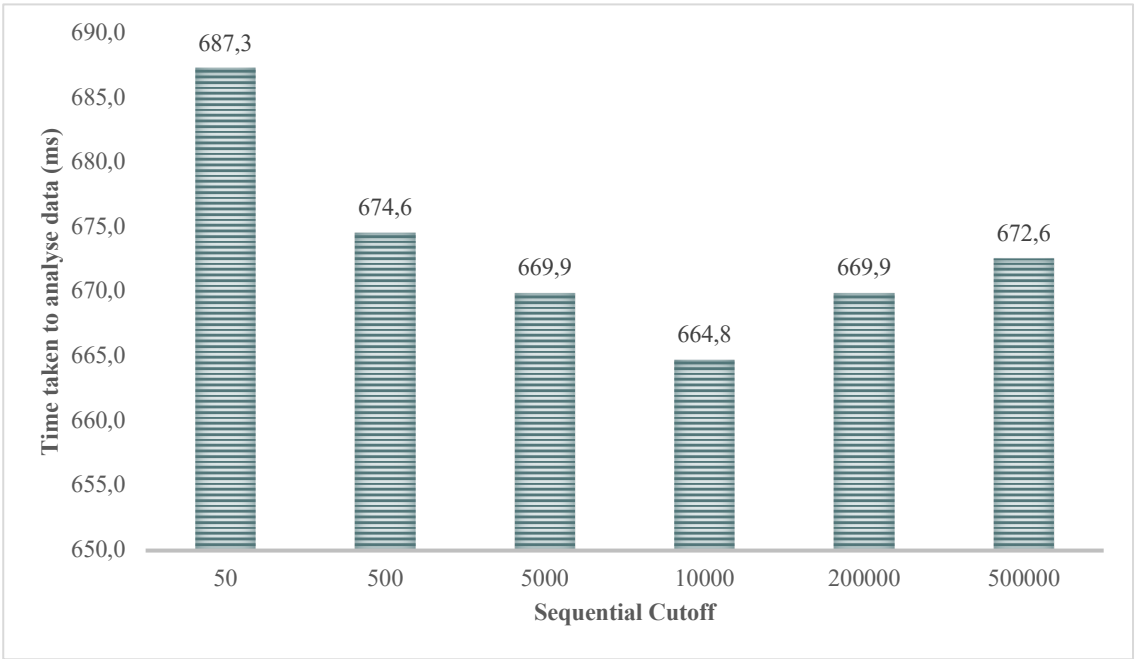

Figure 2: Graph showing time taken to analyse data (milliseconds) for various values of sequential cut-off, using the full data size

However it must be noted that the optimal value of sequential cut-off is highly dependent on both the data size and machine architecture used. Thus the above cutoff value is valid for the full data size and dual-core 2.3GHz Intel Core i5 processor, but may differ given other input data sizes and machine architectures. This is further explored in the following section.

Distinctive Machine Architectures

Data Size

By investigating the four different machine architectures, the following results were obtained, indicating the run times of the parallel algorithm for each architecture with respect to increasing data size.

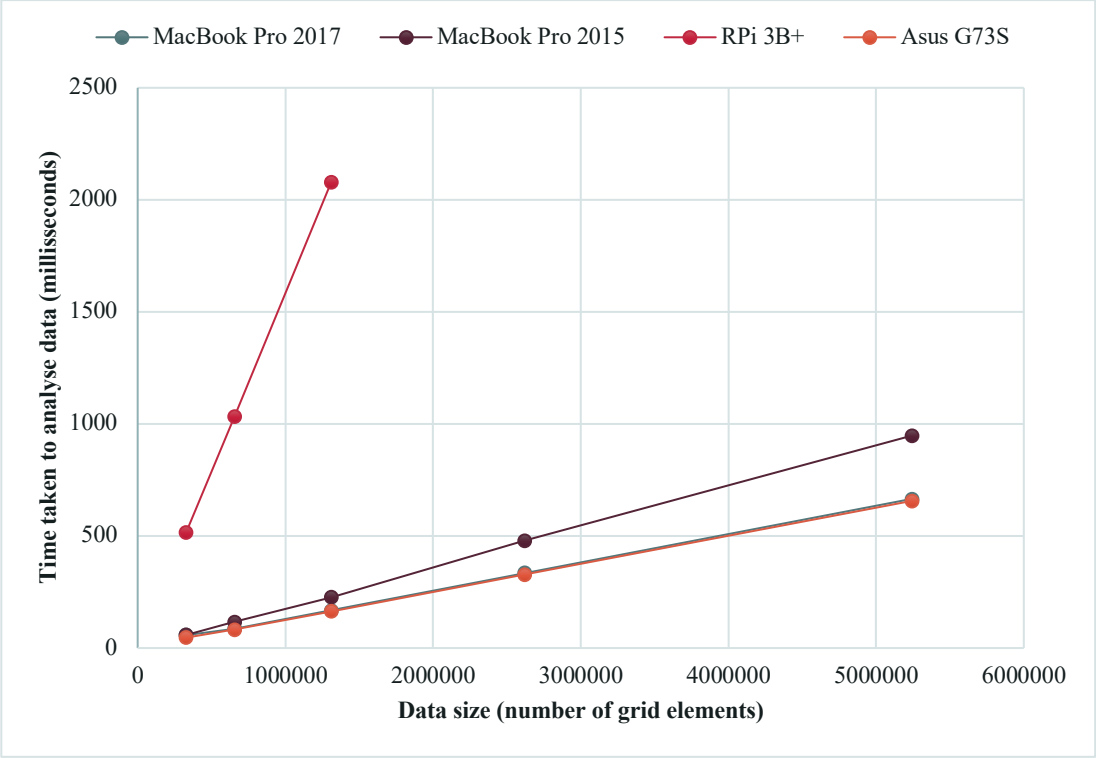

Figure 3: Graph showing the run times of the parallel algorithm with respect to data size, for four different machine architectures

The results indicate that the Asus G73S and MacBook Pro 2017 gave the best performance, with very closely matching data points, followed by the MacBook Pro 2015. This is an interesting result since the MacBook Pro 2015 processor has a faster clock speed of 2.7GHz when compared with that of the MacBook Pro 2017 which has a clock speed of 2.3GHz. However, although both machines house an Intel Core i5 processor, the different versions make a significant impact on the overall performance of each machine. Hence since the MacBook Pro 2017 has a newer generation Core i5 Processor, it indicates better performance and thus one of the best speed up of all the tested machine architectures.

A further interesting observation is the significantly slower run time of the Raspberry Pi 3B+. This is an unexpected result due to the fact that the Raspberry Pi houses a quad-core processor, in comparison to the dual core computers. However, despite the increased number of cores, the Raspberry Pi indicates the worst performance. This is most likely due to decreased clock speed as the Raspberry Pi has a clock speed of only 1.4-GHz in comparison to the 2.3GHz of the MacBook Pro 2017. Another limitation of

the Raspberry Pi includes limited memory space, thus allowing only a quarter of the full input data file to be tested.

Lastly, it is significant that the quad-core i7 processor belonging to the Asus performed largely equally to the dual-core i5 core of the MacBook Pro 2017. Once again this may be due to the version of the respective processors and the year in which they were made. Thus, despite the Asus having twice the number of processors as the MacBook Pro 2017, the Asus performs only marginally better due to differing generations of their respective Intel cores.

Optimal Thread Count

In order to determine the optimal number of threads on each architecture, the run times for varying values of sequential cut-off were analysed. The following figure shows the run times for each architecture for a full data size. It should be noted that the Raspberry Pi is excluded due to its limited memory capacity and hence inability to process the full data size.

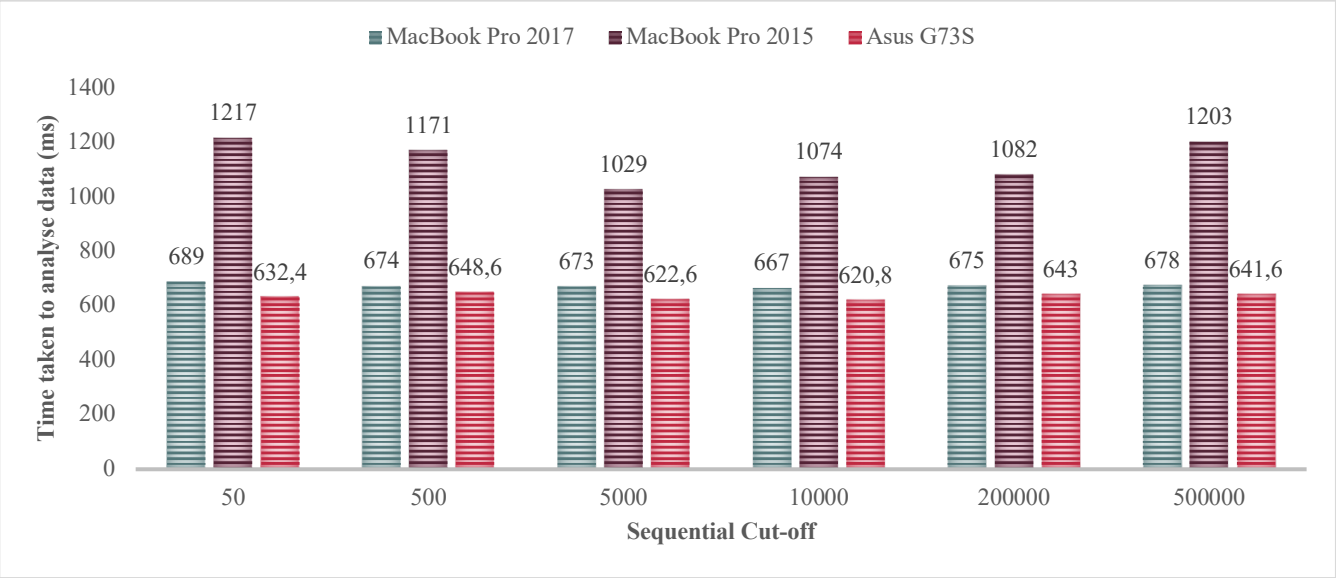

Figure 4: Graph showing the run-time of the parallel algorithm for different sequential cut-off values, for different machine architectures and full data size

It can be observed that the optimal sequential cut-offs are 10 000, 5000 and 10 000 for the MacBook Pro 2017, MacBook Pro 2015 and Asus G73S respectively. This corresponds to an optimal thread count of **7161** threads for the MacBook 2017, **14326** threads for the MacBook 2015 and **7161** threads for the Asus G73S for an input data size of 5242880 grid elements. Thus it can be seen that the optimal thread count is highly dependent on the specific machine architecture.

In order to provide a comparison for the Raspberry Pi, the MacBook Pro 2017 and Raspberry Pi 3B+ were tested with different sequential cut-offs for a quarter of the full data size, giving the following results.

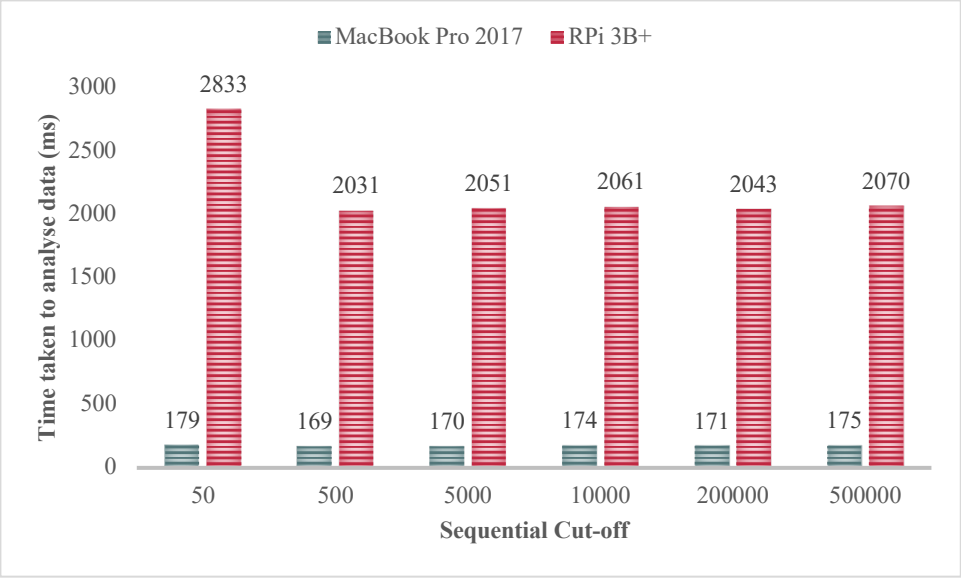

Figure 5: Graph showing the run-time of the parallel algorithm for different machine architectures and a ¼ of the data size

It can be observed that the optimal sequential cut-off for both the Raspberry Pi and MacBook 2017 is 500, for an input data size of 1310720 grid elements. Hence the influence of the data size on the optimal cut-off is demonstrated, as the best cut-off value for the MacBook has changed from 10 000 to 500 due to the change in input data size. The corresponding optimum thread counts is thus **28664** for the Raspberry Pi and MacBook 2017 respectively.

Increased Local Average Area

In order to perform further experimentation, the effect of increasing the area over which the local average wind direction is calculated was investigated. The area was increased from the initial area of the 8 neighbouring units, to 24 neighbouring units and finally, 48 neighbouring units. The results can be seen in the graphs plotting sequential runtimes and parallel runtimes below.

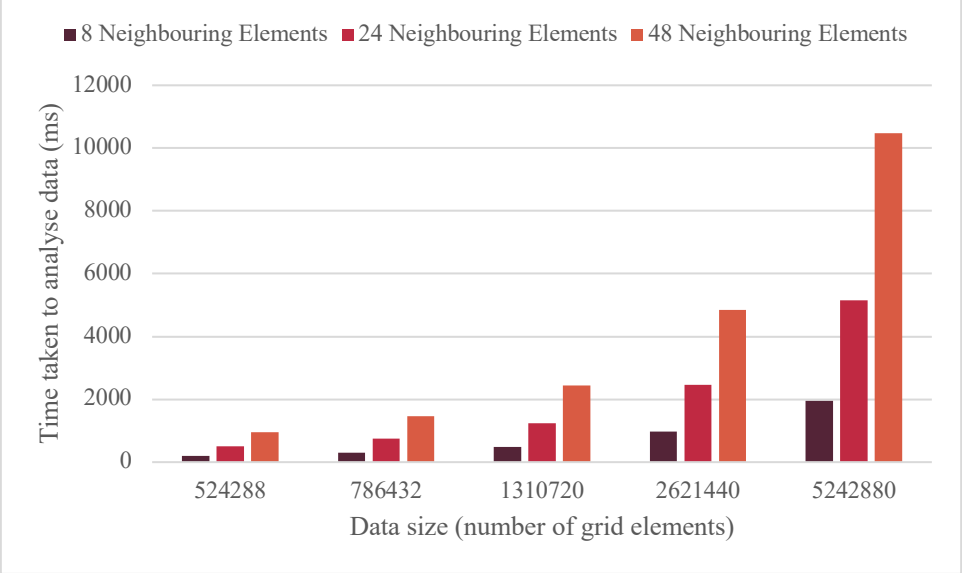

*Figure 6: Figure showing the **sequential** run-time with respect to data size, depending on the area over which local wind is calculated*

It can be observed that the run-time for the sequential algorithm increased significantly for each increase in local range. This is expected as increasing the local average area effectively increases the sequential portion of the algorithm as more elements are iterated over and included in the averaging calculation.

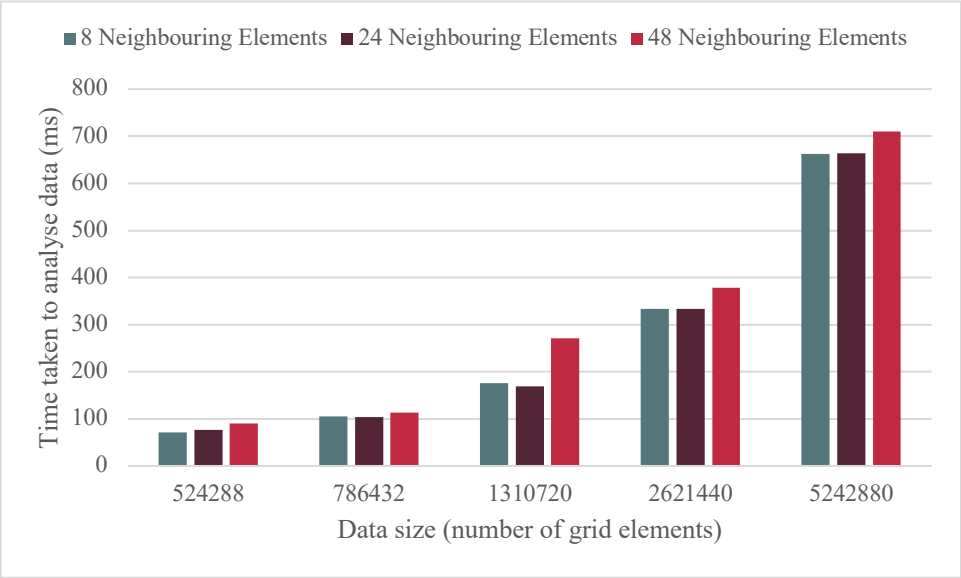

*Figure 7: Figure showing the **parallel** run-time with respect to data size, depending on the range over which local wind is calculated*

In comparison, the run-time of the parallel algorithm increased only marginally for each increase in local range. However, the parallel algorithm was still affected by the larger range, as expected, as an increasing sequential portion of the algorithm causes decreased parallel performance. Hence increasing the area over which the local wind is calculated causes poorer performance for both the sequential and parallel algorithms. The accuracy of the results was found to be unaffected.

Discussion

From the above results, it can be concluded that using parallelization to solve the weather simulation task is worthwhile as it results in an execution time almost three times as fast as the sequential version. This speed up would be increasingly important for larger values of input data size n. The parallel program performed well for a range of data sizes, from 524288 grid elements to 5242880 grid elements, and would be projected to continue good performance for larger values of n. However, for a much smaller value of input data, the overhead of the parallel program outweighs the advantage of parallelization and thus for input data sizes of **524288 grid elements or smaller**, parallelization of the problem is not advantageous.

In order to find the maximum speed up, the optimal combination of parameters was determined. Thus, the maximum speed up obtained using the parallel algorithm was found to be **3,44,** for the full input data size of 5242880 grid elements and a sequential cut-off of 10 000. This value was found to be larger than the theoretical value for 2 processors, thus indicating that the sequential version may include inefficiencies causing the speed up to appear greater.

The optimal sequential cut-off was found to be **10 000** for the full data size and MacBook Pro 2017. It should be noted that this varied based on the given architecture with the optimal thread count being

7161 threads for the Asus G73S and MacBook Pro 2017 and **14326 threads** for the MacBook 2015. Hence the MacBook 2015 required approximately twice the optimal thread count of the other machine architectures. The Raspberry Pi 3B+ was unable to be compared for the full data size.

Conclusions

From the above experiment, it has been established that it is worthwhile to parallelize a computationally heavy task such as weather simulation, as speed up can be attained. For a 2017 dual-core Intel i5 processor, this speed up is approximately **3.44**, for an input data size of 5242880 grid elements. Due to the trend of increasing speed up with increasing data size, it is suggested that the value of speed up will only increase for larger data sizes. Hence for computational tasks requiring the processing of vast amounts of data, parallelization provides a viable solution.

However, the speed up obtained is dependent on a large number of factors including data size, the value of sequential cut-off and machine architecture. It must be noted that for data sizes below a certain value, 524288 grid elements in this case, the difference in run-time between the parallel and sequential algorithms is minimal and thus there is a limited range over which the parallel version has superior performance.

Additionally, the optimal number of threads and thus the optimal sequential cutoff value was found to vary with the particular machine architecture used. Hence it is concluded that in all parallelization problems, particular care must be given to the architecture used and its specific characteristics, in order for the cut-off parameter to be optimized. This is significant in order to ensure the overhead involved in creating the additional threads does not overwhelm the value of extra division of work.

A central extended observation included the importance of memory limitations and clock speed when considering the different available processors for parallel algorithms. This was shown by the Raspberry Pi which exhibited far slower performance due to its reduced clock speed as well as being unable to process large data sizes. A further extension included the observation that by increasing the range over which the local wind was averaged, the sequential component of the algorithm was increased and thus the performance of both the sequential and parallel algorithms was negatively affected. However, the parallel algorithm experienced a far lesser effect than the sequential version, as expected.

The above results are fairly reliable as averages were taken to ensure that the reported values were as accurate as possible, and many sets for each parameter were measured. However, since the results are very specific to a particular architecture, the results would not be considered accurate for a large range of testing conditions and hence may not be widely significant.

Appendix

User Guide

If the program has not been compiled, navigate to the root directory and use *make* to generate the necessary class files. From the root directory, javadocs can be generated and removed using the *make docs* and *make cleandocs* commands respectively. The class files can also be removed using *make clean*. The desired input data file and correct output file should be placed in the *bin* directory. A output file *out_example.txt* has been included in the bin folder for reference.

In order to run the program with the basic functionality, the following command is run from within the *bin* folder:

java CloudData [input data file] [output data file] [correct output data file]

In order to run the program in benchmarking mode, which tests both sequential and parallel, as well as varying the data size and sequential cut-off parameters, the command is adjusted by adding the *-t* flag:

java CloudData [input data file] [output data file] [correct output data file] -t

Git Usage Log

```
Samanthas-MacBook-Pro-3:A3 samanthaball$ git log | (ln=1; while read l; do echo $ln\: $l; ln=$((ln+1)); done) | (head -10; echo ...; tail -10)
1: commit a122ac768953c33f8d381c36a9a1e01988634987
2: Author: Samantha Ball <samanthaball@Samanthas-MacBook-Pro-3.local>
3: Date: Sat Sep 7 20:50:36 2019 +0200
4:
5: added sample input and output files for reference
6:
7: commit 02b805ae4db01576dbb98eec5af8678e84fd3c06
8: Author: Samantha Ball <samanthaball@Samanthas-MacBook-Pro-3.local>
9: Date: Sat Sep 7 20:17:21 2019 +0200
10:
...
122: Author: Samantha Ball <samanthaball@Samanthas-MacBook-Pro-2.local>
123: Date: Sun Aug 25 13:43:58 2019 +0200
124:
125: created boundsCheck method for use in local wind calculation
126:
127: commit 4fe05fd250f0689e0919e316c559c09ad7d85f54
128: Author: Samantha Ball <samanthaball@Samanthas-MacBook-Pro-2.local>
129: Date: Sun Aug 25 13:40:21 2019 +0200
130:
131: fixed issues with Vector class
```

Git repo: https://github.com/SamBall999/CSC2002S-A3