CSC2002S Assignment 3

By Leanne January (JNRLEA001)

Table of Contents

Introduction

In this project, I shall be writing a program to process data for a single layer of air as it evolves over time in order to predict the prevailing wind over the simulation and the types of clouds that might form as a result.

Project Aim

The aim of this project is to process the data in a way that is correct and to find the optimal way to parallelise the data in order to reduce the time it takes for the program to complete, thereby increasing its performance. To do this, I will be implementing the Java Fork/Join framework.

Parallel Algorithms & Expected Speedup/Performance

The parallel algorithm I will be using is the Divide and Conquer Algorithm. The Divide and Conquer Algorithm lets threads dynamically spawn other threads until each thread has a reasonable amount of work to do. The amount of work a thread does is determined by the sequential cut-off which is used to tell the program when to stop creating new threads and let the current threads each execute a set of sequential code which contains the actual calculation or processing of data. This, therefore, allows for multiple threads to handle the calculations for different sets of the data at the same time instead of one thread iterating through all the data elements doing each calculation one by one.

I have chosen to do the Divide and Conquer Algorithm as the different calculations I will need to do are all embarrassingly parallel – meaning that the computation can be divided into completely separate parts. The Divide and Conquer method is best used for embarrassingly parallel problems as it is the most efficient way to divide up tasks and complete them.

The expected speedup of a Divide and Conquer Algorithm is O (log n) meaning that it has exponential performance growth. However, the only time exponential performance growth is actually achieved is if the number of threads the program equals the number of cores or logical processors. This means that for a program with 1000 threads to come close to having perfect exponential speedup, each thread must be run on a single core – therefore, there would need to be 1000 cores available. This is because creating more than a single thread on one core introduces additional time costs called overheads which factor in the time taken to create a thread and dispose of it.

Methods

Parallel Solution: Calculating the Prevailing Wind

To calculate the prevailing wind in parallel, I implemented RecursiveTask<T>. RecursiveTask allows a thread to return a value, in this case an object of the Vector class.

The compute method, which returns the prevailing wind, either instructs a thread to add up all the x and y components for all the vectors in a given range, find the average vector and return its magnitude, or it instructs a thread to add two vector magnitudes together.

This is an example of a Reduce pattern as each reduction operation produces a single answer (a Vector object) from a collection via an associative operator.

Parallel Solution: Classifying the Clouds

To classify the clouds in parallel, I implemented RecursiveAction. RecursiveAction does not have a return type, and therefore allows a thread to perform a task without having to return a value.

The compute method allows a thread a set of elements, for each of which it has to calculate the local average and using this, predict what type of cloud would form. The type of cloud is represented by an integer value which the thread assigns to an integer array inside the CloudData class which keeps track of what types of cloud form at each specific air location.

This is an example of a Map pattern as each element of the collection needs to be operated on independently to find the local average wind direction and the type of cloud.

Testing Methods

Correctness

To test for correctness, I made use of the Ubuntu command `diff file1 file2` to check if my large output matched the given sample output.

Timing

I implemented the tick() and tock() methods in the Main class to measure the time it takes for the two parallel solutions to complete, ignoring the time it takes to read in the data from the disk and write to files.

Measuring speedup

Java Warm-Up

In order for the program to achieve its top potential speed I repeated the timing of the experiment within the program, threw away the first two

results and took an average of the next 8. This works as the Java Virtual Machine stores the data in cache once it has been accessed by the programme. This reduces the speed as the programme can then access the data from cache instead of from RAM.

Number of Threads/Sequential Cut-off

Number of Threads = N = Number of threads in the middle of the DAG for the Divide and Conquer Algorithm.

The sequential cut-off is the point where the program stops acting in parallel by stopping creating new threads. N threads perform the sequential calculation of small subsets of the data simultaneously. In order to achieve the best performance, the optimal sequential cut-off where one has optimal N and each thread is doing an optimal amount of work, must be found. To find the optimal sequential cut-off, a range of many different sequential cut-offs must be tested, and their speedups compared.

The sequential cut-off, however, is also dependant on the size of the data, as a larger data size will need a larger sequential cut-off to yield the same N. In order to be constant throughout the experiment, each data size will need to have the same N or range of N, and the sequential cut-offs for each N in each data size will need to be determined.

Firstly, I created a range of N from N1=1 to N20=524 288. Each value for N goes up in increasing order of 2 starting from 0 so that N1 = 1, N2 = 2, N3 = 4, N4 = 8 ... N20 = 524 288.
Secondly, for each data size I used to following calculation to determine what the sequential cut-off needed to be for each N:

Data size / N= Sequential Cut-off

So, for a data size of 5 000 000: if I would like to test how fast 128 N are I would need to input a sequential cut-off of 78 125.

This gave me a sequential cut-off for each N which I used to test for speedup vs number of threads (N)

Measuring machine architectures

In order to find the optimal number of cores needed for the best parallel performance, I tested this program on three different architectures. One with one core, one with two cores and one with three cores.

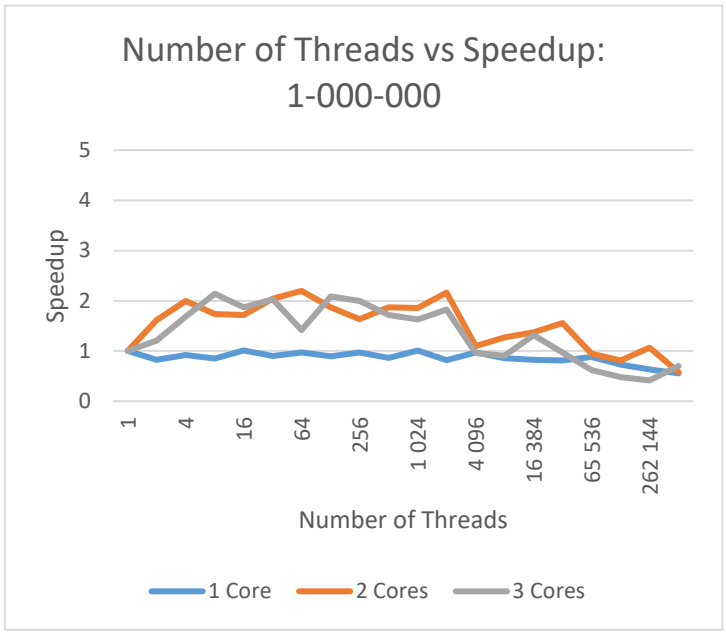
Problems and Difficulties

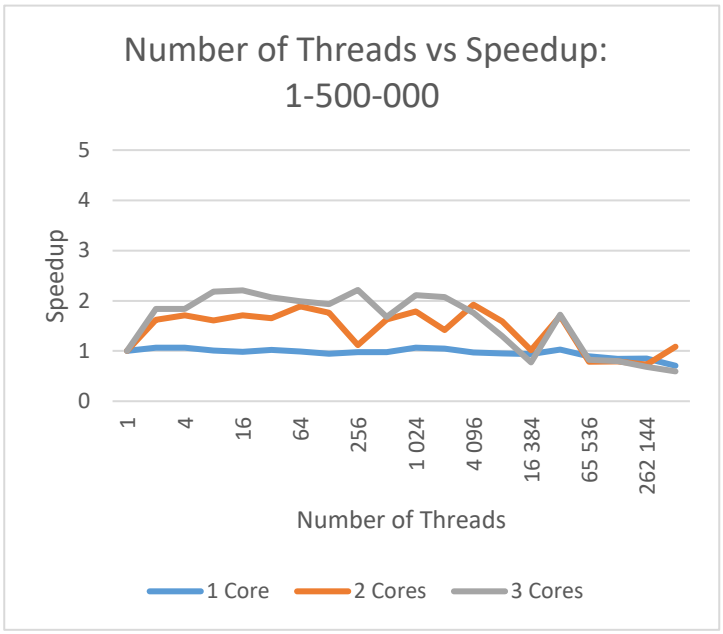
Java Memory Constrains

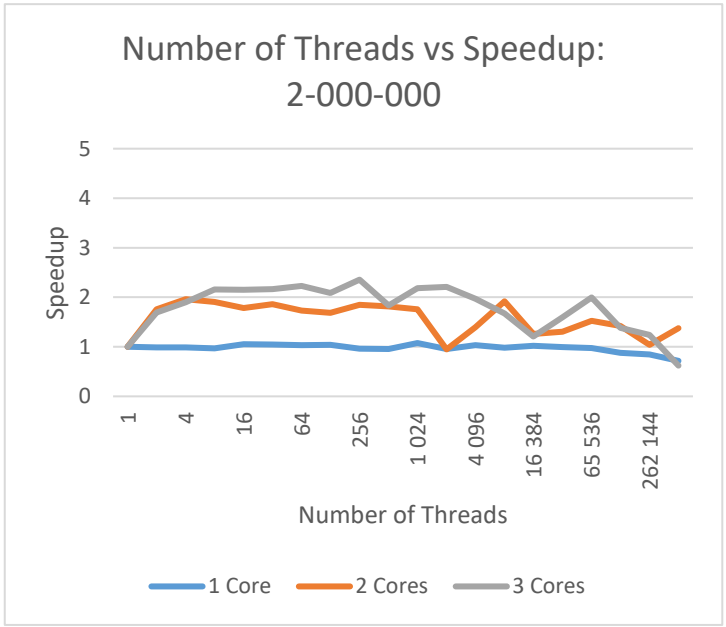
My programme had errors when inserting files with a data size of over 6 000 000. This constrained the range of the data sizes to 1 000 000 – 5 500 000. I would have preferred the data size to range from 5 000 000 to 50 000 000 to test the performance of the program and very large data sizes.

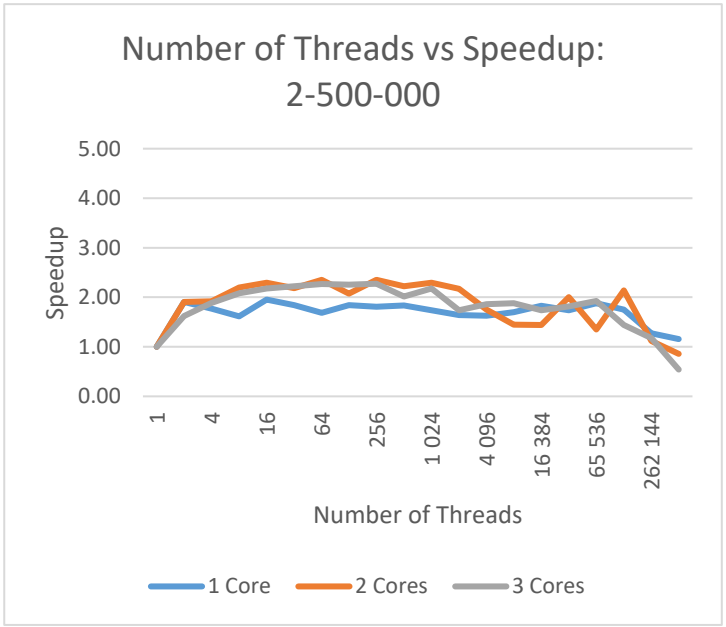
Results and Discussion

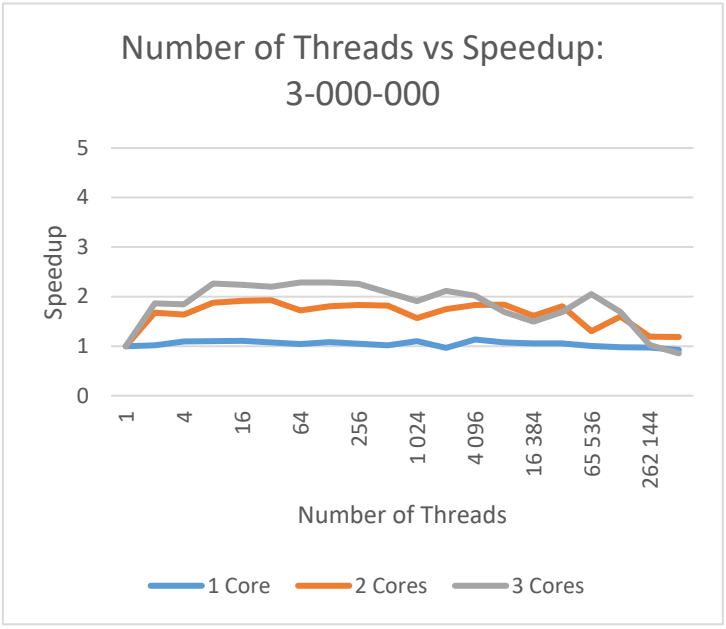
Graphs depicting the Number of Threads (N) vs Speedup for different data sizes

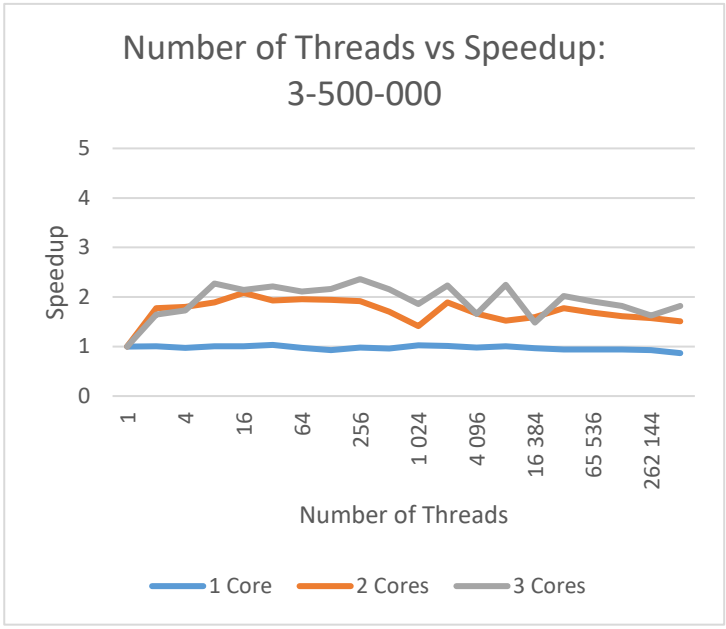


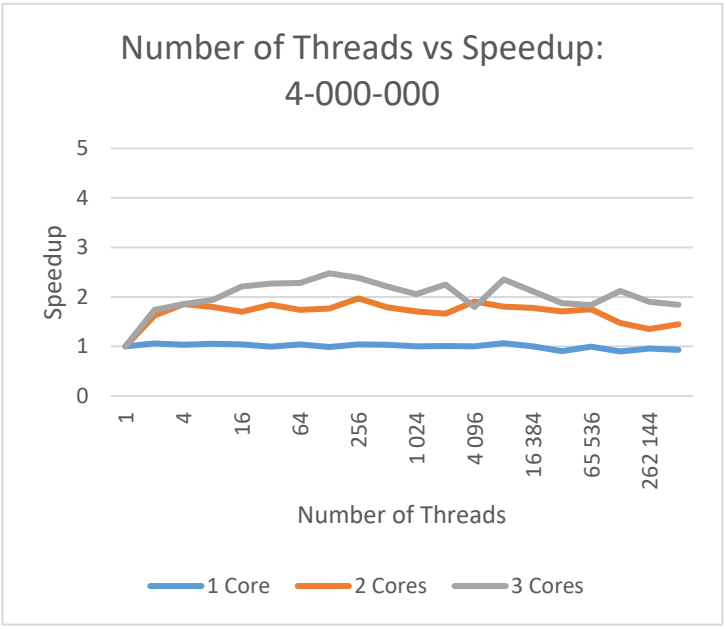


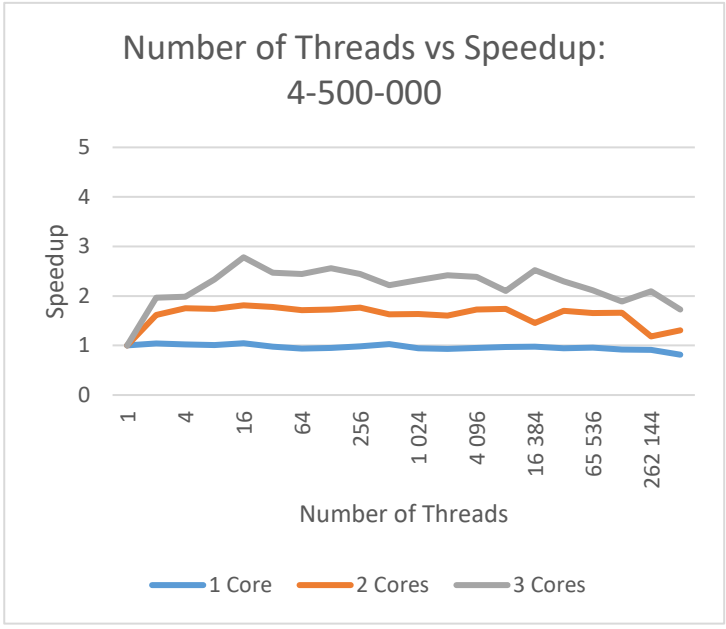


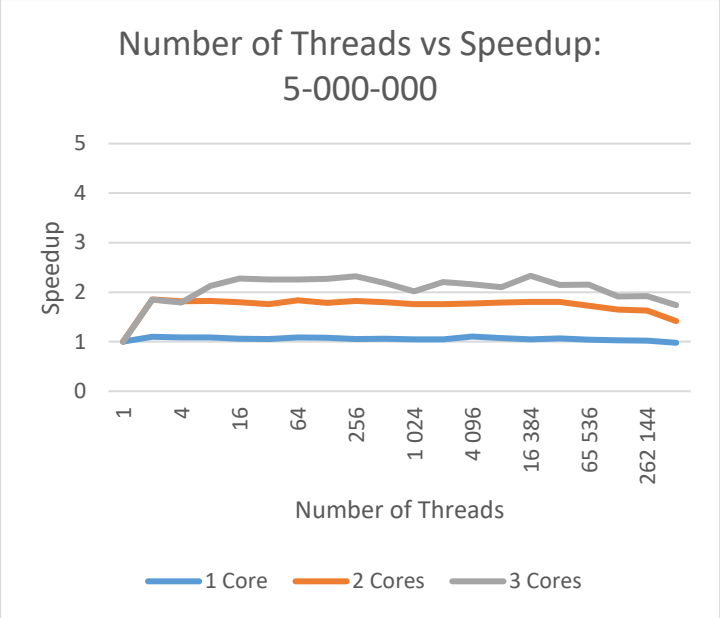


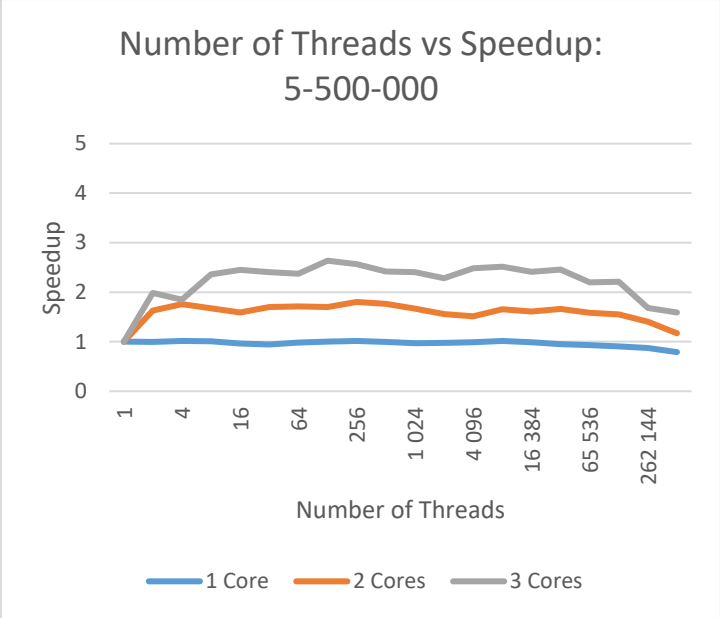

Number of Threads vs Speedup: 3-000-000
Speedup
0
1
2
3
4
5
1
4
16
64
256
1 024
4 096
16 384
65 536
262 144
Number of Threads
1 Core
2 Cores
3 Cores


Number of Threads vs Speedup: 3-500-000
Speedup
0
1
2
3
4
5
1
4
16
64
256
1 024
4 096
16 384
65 536
262 144
Number of Threads
1 Core
2 Cores
3 Cores


Number of Threads vs Speedup: 4-000-000
Speedup
0
1
2
3
4
5
1
4
16
64
256
1 024
4 096
16 384
65 536
262 144
Number of Threads
1 Core
2 Cores
3 Cores


Number of Threads vs Speedup: 4-500-000
Speedup
0
1
2
3
4
5
1
4
16
64
256
1 024
4 096
16 384
65 536
262 144
Number of Threads
1 Core
2 Cores
3 Cores



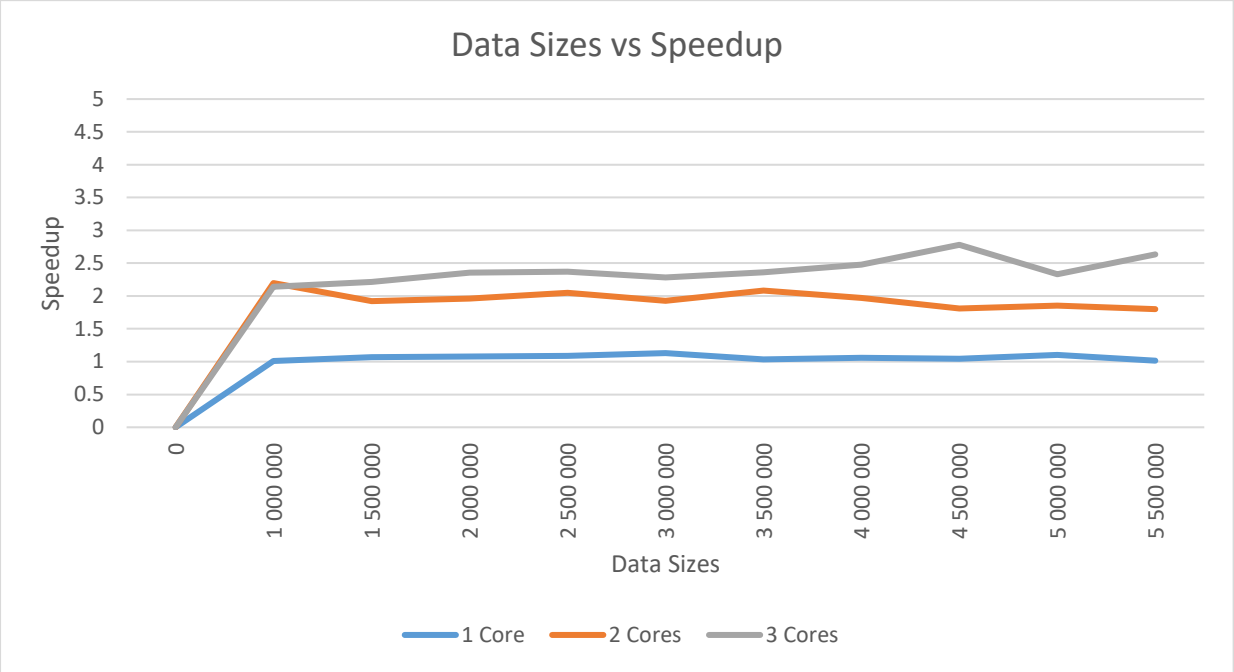


Observations

1. Executing the program on one core consistently yields the worst performance. The speedup is, in almost every case, a straight line of y = 1 which tends to a negative gradient as the data size gets enormously large.

2. The shape of the speedup for the two and three core architectures increases with a line of x = y at first, before straightening to a gradient of 0 and then decreasing as the number of threads (N) becomes extremely high. The speedup of the three core architecture does tend to a more parabolic shape as the data size increases. The decrease in speedup towards a large N is also expected as there are many overheads to consider with an N that large.

Graph depicting the <u>Data Size vs Speedup</u>



Observations

1. As the data size increases, running the program in parallel on a one core architecture has no performance increase - its line has a gradient of 0 and towards the end it starts to decrease with a gradient that is the most steeply negative. One core architectures are more suited to running programs sequentially as there is only one logical processor to create threads from. As soon as more than one thread is created the overheads start to impact performance levels, and as data size increases the amount of threads needed to be effective are outweighed by the amount of overheads they incur.

2. The two core architecture starts out with its highest performance and as the data size increases, its speedup gradually decreases. The reason is also because the overheads needed to create enough threads impacts heavily on the speedup of the program for large data sizes. This also may just be a random dip in results and we can still expect the speedup to increase with larger data sizes. Running the program in parallel on a two core architecture is also still more or less twice as fast as running it sequentially.

3. The three core architecture increases the speed of the program gradually over the data sizes less than 5 000 000, but as soon as the data size is greater than 5 000 000 the program has a steep and positive gradient and can be expected to do well with very high data sizes.

Discussion

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

The above graph shows there is a significant speedup to be gained from using parallelisation in Java. The graph also indicates that there is a high speedup for very large sizes of data and weather simulation and prediction does call for massive amounts of data. It may be worth simulating relatively small weather predictions in Java (such as the one for this project), but clusters may be suited better to very large simulations with incredibly massive amounts of data.

The range of data set sizes for which my parallel program performs well

The parallel program only has a gradual increase in performance with data set sizes of less than 4 500 000. The graph indicates that the program may be better suited to very large data set sizes of 40 000 000 – 50 000 000.

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

My program yielded a maximum speedup of 2.78125 for a data set size of 4 500 000 on a 3 core machine architecture. The speedup for 3 processors:

T1/T(3) = Speedup

Where T1 is the time for 1 processor and T3 is the time for P=3 processors.

Therefore,

328.375ms/136ms= speedup of 2.415

This means that my maximum speedup is greater than the theoretical speedup.

What is an optimal sequential cut off for this problem?

The sequential cut-off depends on the number of threads and the size of the data set. From the graphs, the highest speedups occur in the range of 16 – 256 number of threads. For a data set size of 5 500 000, the optimal sequential cut off would be between 85 938 and 171 875, therefore 128 906.

What is the optimal number of threads on each architecture?

For an architecture with one core, the optimal number of threads is theoretically 1 and realistically 1 as the performance of the parallel program on one core was not any greater than the sequential performance.

For an architecture with two cores, the optimal number of threads is theoretically 2 but realistically it tends to be either 4 or 16.

For an architecture with three cores, the optimal number of threads is theoretically 3 but realistically the optimal number of threads is generally between 64 and 256, tending to 256 as the data set size increases.

Final Conclusions

1. The more logical processors one has, the better one is able to use parallelism to increase performance.

2. Parallel programs have higher speedup for large data sizes, and they tend to not do well with smaller data sizes.