CSC2002S Assignment 3

Callum Tilbury

September 12, 2019

Contents

1 Introduction

Weather simulations and predictions are an important tool for society for a host of reasons – safety being the most critical one. However, such simulations can easily become terribly complex due to the sheer amount of data in them. And yet, such simulations are useless if they cannot be performed quickly and reliably, before the predicted event actually happens. This requirement of speed, coupled with the nature of the problem – lots of repeated "number crunching" – makes parallel processing an ideal solution.

The aim of this particular weather-themed problem involved the averaging and classification of wind data over time, and over a planar area in space. Grids of data for one or more time steps were provided, with each containing multiple points, and each point contained the wind-information at that particular point in space and time: an advection vector (the direction of the wind parallel to the plane), and the convection component (the direction of the wind normal to the plane). Two values were calculated from this data: the *prevailing wind average* (which was the average advection vector over all space and all time), as well as the *cloud classifcation* (which was the predicted cloud type that will result at a specific point, at a specific time).

The parallel algorithms used in this problem fall into two distinct patterns: *reduce*, and *map*. The former describes the prevailing wind average calculation – which used a divide-and-conquer approach by splitting the data into many subsets. The latter describes the classification calculation – which used a localised average that mapped to a separate classification grid, with each localised region independent of the others. The work (T_1) and span (T_∞) of these patterns are given below:

$$\begin{aligned} \text{Prevailing Average: } T_1 = O(n) \quad ; \quad T_\infty = O(\log n) \\ \text{Classification: } T_1 = O(n) \quad ; \quad T_\infty = O(\log n) \end{aligned}$$

Hence, the ideal speed-up is $O(\frac{n}{\log n})$. Of course, the actual expected speed-up achieved by the parallel algorithm is severely limited by the finite number of processors available, and is much lower than the ideal value.

2 Method

2.1 Overview

There are usually many ways in which one can approach a problem, and this situation was no different; the task at hand could be performed in multiple variations of loops and sums. Even within a particular algorithm pattern, there were many choices. The challenge, then, was to find an technique that most appropriately suited the context, and was moreover suitable for *parallelization*. For example, one could have embraced a divide-and-conquer approach by splitting the two-dimensional grid into four equally-sized smaller grids. However, using threads for this approach did not scale well with datasets that had many time-steps recorded, as each grid in time still had to be operated on independently. There was an inherent sequential nature to such an approach, and its parallel performance was thus adversely affected. Instead, a suitably parallelizable algorithm was chosen, and is explained in the sections that follow.

In terms of implementation, the parallel algorithms were written using the Java Fork/Join framework; specifically, the RecursiveTask type was inherited. To facilitate varied testing, code was written to generate files of random floating point numbers (between 0 and 10) of a given size. One could specify the number of time steps t, rows r and columns c. For each set of tests such a file was created.

When assessing the speedup, it was important to change only one variable at a time. Failing to do this may have resulted in incorrect assumptions. Thus, for each of the experiments, as much as possible, all factors (t, r, c, and the sequential cutoff) were kept constant except one.

2.2 Data Linearisation

The chosen algorithm and its designed methods rely on a fundamental requirement: the data should be in a single, linear array of data, containing the information about the wind over all time and all space. In the input file, the data is a two-dimensional grid with a third-dimension of time, and each element in the grid is a vector with three components (x, y, u – the horizontal and vertical advection components, and the convection component respectively). It was thus required for this data to be converted to a single linear array, as depicted in the example in figure 1. The resulting array had the form:

$$[x_{0:00} \quad y_{0:00} \quad u_{0:00} \quad \dots \quad x_{t:rc} \quad y_{t:rc} \quad u_{t:rc} \quad \dots \quad x_{\dim(t):\dim(r)\dim(c)} \quad y_{\dim(t):\dim(r)\dim(c)} \quad u_{\dim(t):\dim(r)\dim(c)}]$$

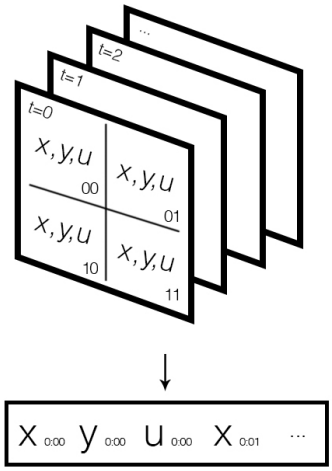


Figure 1: Graphical example of the linearisation process

2.3 Prevailing Wind Average

The prevailing wind average, denoted $\bar{\boldsymbol{w}}$, gives a general idea of the strength and direction of the wind in a region over a particular time period. It is defined as the average advection vector over all time and all space in a given dataset. That is,

$$\bar{\boldsymbol{w}} := \frac{1}{\dim(t) \cdot \dim(r) \cdot \dim(c)} \left(\sum_{t,r,c} x(t,r,c) \quad , \quad \sum_{t,r,c} y(t,r,c) \right)$$

where $x(t,r,c)$ and $y(t,r,c)$ are the two components of the advection vector at a point, $\boldsymbol{w}(t,r,c)$.

Using the linearised dataset discussed in section 2.2, calculating the sum (and thus average) was trivial. For the sum of x-components, the code started at index [0], and simply added every third element. The y-components were done similarly, except starting at index [1].

This process was easily parallelizable. A divide-and-conquer approach was taken, where different threads received indices of different subsets of the full array of data. Each thread simply added the contributions of *that* subset to the overall sum. When all the threads were finished, the overall sum was divided by the total number of elements (a known figure) to find the prevailing wind average values.

Note that in order to ensure every subset started on an x-element, the indices of the upper and lower bounds were shifted by the following operations:

$$\text{upper}^* = \text{upper} - (\text{upper} \ \% \ 3)$$

$$\text{lower}^* = \text{lower} - (\text{lower} \ \% \ 3)$$

Pseudocode for calculating the prevailing wind average in parallel – using the compute(·) method in the Java Fork/Join framework – is given in listing 1:

Listing 1: Pseudocode for calculating the prevailing wind average

```
compute() {
  if((hi-lo)<SEQUENTIAL_CUTOFF) {
    // Shifts subset down to start on an x-element
    lo = (lo-lo%3);
    hi = (hi-hi%3);

    // Summing operations
    float sumX = 0, sumY = 0;
    for(int pt = lo; pt < hi; pt++) {
      // linearSet is the full set of data
      sumX += linearSet[pt++];
      sumY += linearSet[pt++];
    }

    return new Result(sumX, sumY);

  }

  else {
      // Split into two subarrays
      left = new Sum(linearSet, lo, (hi+lo)/2);
      right = new Sum(linearSet, (hi+lo)/2, hi);

```

```
24        // Parallel step
25        left.fork();
26        Result rightAns = right.compute();
27        Result leftAns = left.join();
28
29        return new Result(rightAns.x() + leftAns.x(),
30              rightAns.y() + leftAns.y());
31      }
32    }
33  }
```

2.4 Cloud Classification

Cloud classification is a way of estimating the nature of the cloud that will form at a specific location, at a single point in time, using the weather data at- and surrounding- that point. Consider the point (r, c) in space, at a particular time recording t. Let the advection vector at this point be denoted $\boldsymbol{w}(t, r, c)$, and the convection component $u(t, r, c)$. Define the *local average wind vector* $\hat{\boldsymbol{w}}(t, r, c)$ as the average wind vector of the point and its immediate neighbours (not exceeding the grid's boundaries). That is,

$$\hat{\boldsymbol{w}}(t, r, c) := \frac{\sum_{i=r_{\text{low}}}^{r_{\text{high}}} \sum_{j=c_{\text{low}}}^{c_{\text{high}}} \boldsymbol{w}(t, i, j)}{(c_{\text{high}} - c_{\text{low}} + 1)(r_{\text{high}} - r_{\text{low}} + 1)}$$

where

$$\begin{aligned} r_{\text{low}} &= \max[r - 1, 0] \\ r_{\text{high}} &= \min[r + 1, \dim(r)] \\ c_{\text{low}} &= \max[c - 1, 0] \\ c_{\text{high}} &= \min[c + 1, \dim(c)] \end{aligned}$$

Using these definitions, a simplified classification model was used in the experiment, and is summarised in the equation below:

$$\text{classification}(t, r, c) = \begin{cases} 0 & |u(t, r, c)| > |\hat{\boldsymbol{w}}(t, r, c)| = \sqrt{\hat{w}_x^2(t, r, c) + \hat{w}_y^2(t, r, c)} \\ 1 & |u(t, r, c)| \leq |\hat{\boldsymbol{w}}(t, r, c)| \cap |\hat{\boldsymbol{w}}(t, r, c)| > 0.2 \\ 2 & \text{otherwise} \end{cases}$$

where the classifications of 0, 1 and 2 correspond to cumulus, striated stratus, and amorphous stratus clouds respectively.

The primary computation required in this cloud classification challenge is the sum of the wind vectors at a point and its immediate neighbours. There are at least two viable ways of performing this localised sum, here termed *implosive* and *explosive* methods. Figure 2 shows graphically the distinction between the two.

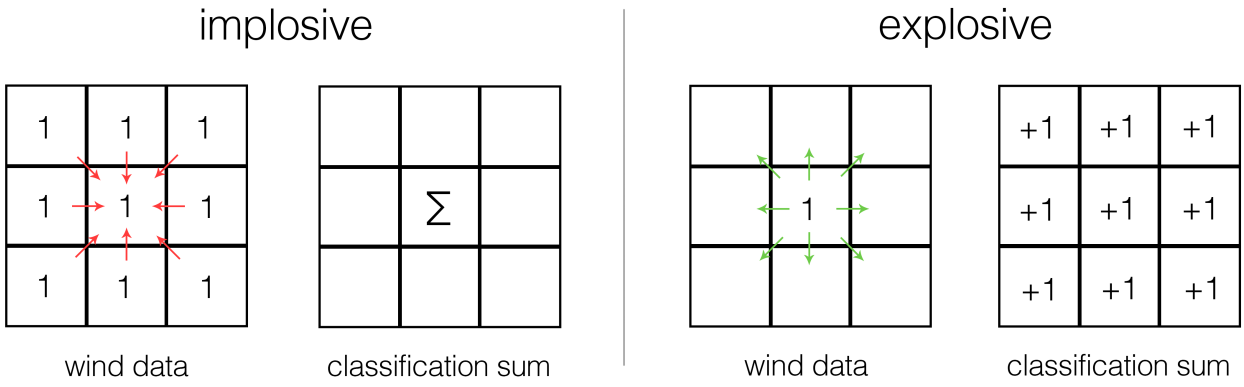


Figure 2: Graphical example showing the difference between implosive and explosive localised summing methods

In code, for both cases, a grid for the "classification sum" data must be created. The wind data is then traversed and each point (t, r, c) is visited once. When using the implosive method, the program puts the sum of the wind vectors of the current point and its immediate neighbours in the classification sum grid at that point (t, r, c). When using the explosive method, on the other hand, the program adds the wind vector of current point to the classification sum data at that point, as well as at the points of all the immediate neighbours. Importantly, once the entire wind data grid has been traversed, the classification sum will be identical, regardless of the method.

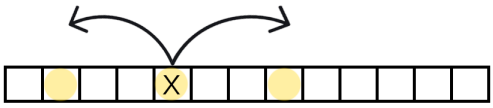


Figure 3: Visualisation of the explosive method with the linearised dataset

The subtle difference with the explosive method is advantageous when dividing up the linearised dataset. Consider figure 3.

Notice that the explosive method requires no knowledge of the other subsets of data, whereas the implosive method may need to bring in data from other parts of the larger array. Provided the explosive method can identify the time and place corresponding to its current array position, it can successfully push its own value to the correct sections of the classification sum array. Moreover, since the prevailing average algorithm (see section 2.3) is already traversing each element in the wind data grid, the classification sum can be incorporated into a single compute(·) method.

Note that, as mentioned, the explosive method *does* need to be able to identify the time and position it currently is in, within the linearised data array, given its array index γ. These values can be calculated as follows:

$$\begin{aligned} t(\gamma) &= \left\lfloor \frac{\gamma}{3 \cdot \dim(r) \cdot \dim(c)} \right\rfloor \\ \text{Let } n(\gamma) &= \frac{1}{3}\gamma - t(\gamma) \cdot \dim(r) \cdot \dim(c) \\ \text{Then } r(\gamma) &= \left\lfloor \frac{n(\gamma)}{\dim(c)} \right\rfloor \\ \text{and } c(\gamma) &= n(\gamma) - r(\gamma) \cdot \dim(c) \end{aligned}$$

The derivation of these equations follow naturally from the structure of the linearised dataset.

The point (t, r, c) then corresponds to the index $[t \cdot \dim(r) \cdot \dim(c) + r \cdot \dim(c) + c]$ in the classification sum array.

Pseudocode for the classification sum algorithm is given in listing 2.

Listing 2: Pseudocode for calculating the classifications in parallel

```
compute() {
  if((hi-lo)<SEQUENTIAL_CUTOFF) {
    // Shifts subset down to start on an x-element
    lo = (lo-lo%3);
    hi = (hi-hi%3);

    int t, n, i, j;
    // Iterate through each point in this subsection
    for(int pt = lo; pt < hi; pt++) {
      t = pt/(3*dimr*dimc);
      n = (pt - (t*3*dimr*dimc))/3;
      i = n/dimc;
      j = n - i*dimc;

      currX = linearSet[pt++];
      currY = linearSet[pt++];

      // Explode this data to surrounding elements
      for (int r = Math.max(0, i-1); r <= Math.min(dimr-1, i+1); r++) {
        for (int c = Math.max(0, j-1); c <= Math.min(dimc-1, j+1); c++) {
          classification_sum[t*dimr*dimc + r*dimc + c].incr(currX, currY);
        }
      }

    }

    else {
      Sum_and_Classify left = new Sum_and_Classify(linearSet, lo, (hi+lo)/2);
      Sum_and_Classify right = new Sum_and_Classify(linearSet, (hi+lo)/2, hi);

      left.fork();
      right.compute();
    }
  }
}
```

2.5 Sequential Cutoff

Parallel algorithms require a parameter called a *sequential cutoff*. This value indicates the smallest unit of work that a single thread performs. In the context of this experiment, this cutoff represents the size of the array (which is a subset of the full dataset) on which a single thread operates. By setting this value to be too large, the algorithm becomes serial in nature – for example, by selecting a sequential cutoff that is equal to the size of the dataset, only a single thread will spawn, and this is essentially equivalent to a serial implementation. In contrast, by setting this value to be too small, too many threads will spawn and the cumulative overhead of the thread creation will result in poor performance.

Thus a 'sweet spot' must be found, and this is highly dependent on the context – such as the parallel algorithm used, the overhead cost of creating a single thread, the size of the dataset, and the architecture on which the code is running.

2.6 Validation

A fast algorithm is useless if it outputs incorrect results, and thus accuracy was an important consideration for all tests. To facilitate this, a Java class was written to read in two files and compare them, character by character, and provide the number of characters that differ in the files. This was required for both results: the prevailing wind average, and the classification output data.

The verification process was actually done in two parts: Firstly, it was important to confirm that the algorithm discussed here agreed with the requirements of the project. This was done using the sample input and output data provided by A/Prof. Gain. The serial code was run on the two sample input files, and the outputs of these were compared to the respective sample output files provided. Once this was done, it was assumed that the serial code was running correctly. Then, for each test thereafter, the serial implementation was used to check the correctness of the parallel implementation.

Note that throughout this practical, 32-bit floating point numbers were used to hold the data. Hence, inevitably, as the data sets became large, the results produced were occasionally inaccurate. This was immediately obvious in the prevailing wind averages – which were often only accurate to around 5 or 6 decimal places. Every now and then there were also a handful of classification errors. Provided these errors remained fairly insignificant – prevailing averages still correct to 4 or 5 decimal places, and the number of classification errors was less than 0.01% of the dataset size – then the algorithm was assumed still to be accurate.

2.7 Timing

In order to establish meaningful metrics in this investigation, a reliable timing facility had to be used. This was done using the system method, *System.currentTimeMillis( )*. This method returns the number of milliseconds since the 'UNIX epoch' – the 1st of January 1970. By recording this value before and after the program executes, and finding the difference between them, the approximate run-time of the program can be found. This is shown in pseudocode in listings 3.

Listing 3: Pseudocode for measuring execution time

```
static long startTime = 0;
private static void tick() { startTime = System.currentTimeMillis(); }
private static float tock() { return (System.currentTimeMillis() - startTime); }

main() {
  /* Untimed activities */

  // Start timing:
  tick();

  /* ------- DO STUFF ------- */

  // Stop timing
  float time = tock(); // Stop timing

  /* Untimed activities */

}
```

Note that since this experiment aimed to compare the performance of the serial and parallel algorithms alone, irrelevant actions were omitted from the timing data. This included all file handling actions (opening, closing, writing, reading), printing to the screen, and setting up of the arrays.

2.8 Speedup

Speedup is defined as the factor by which the code's runtime decreases ("speeds up") when using the parallel algorithm, as opposed to the serial version. In order to prevent anomalous results, each set of test parameters was run several times: the serial code was run 5 times, and the parallel code 10 times. The first result of each of these tests was discarded to account for cache warming. With the remaining data, the average execution time was calculated for serial and parallel algorithms respectively. The speed-up for that test was then calculated as: speed-up $= \frac{\bar{t}_{\text{serial}}}{\bar{t}_{\text{parallel}}}$, where $\bar{t}$ indicates the average execution time.

2.9 Architectures

To understand the effect of computer architecture on the results, all tests were run on two different computers, the details of which are given in table 1. Importantly, notice the varying clockspeeds, as well as the different number of physical cores in each machine.

Device	Nickname	Processor	Physical Cores	Cache	Memory
Apple MacBook Pro 13" (Mid-2015)	MacBook 13"	2.7 GHz Intel Core i5 processor	2	L2: 256KB p/core; L3: 3MB	8GB
UCT CS Nightmare Server Computer	Lab PC	2.4 GHz Intel Xeon CPU E5620	4	12 288 KB	23GiB

Table 1: Table showing the computer architectures used for this practical

3 Results & Discussion

3.1 Dataset Size

An important consideration in the parallel algorithm's performance is the size of the dataset. This is due to *overhead* – the inherent cost involved in setting up threads. If the dataset is small, the benefit of running the algorithm in parallel may be negated by the large overhead costs relative to the total execution time. Essentially, by the time the threads have been set-up and are ready to begin, the serial code may already have finished all the work. The test results in figure 4 explore this idea. For this test, the sequential-cutoff was fixed at 1000 elements, and only a single time-step was used ($t = 1$). For each dataset size, a square grid was used ($r = c$), and the reported 'Number of Data Elements' was the product of the two dimenstions ($r \times c$), indeed giving the total number of elements. Note that the graph uses a logarithmic scale for the independent variable, as the effect of speed-up only changes noticeably with an order of magnitude change in the dataset size.

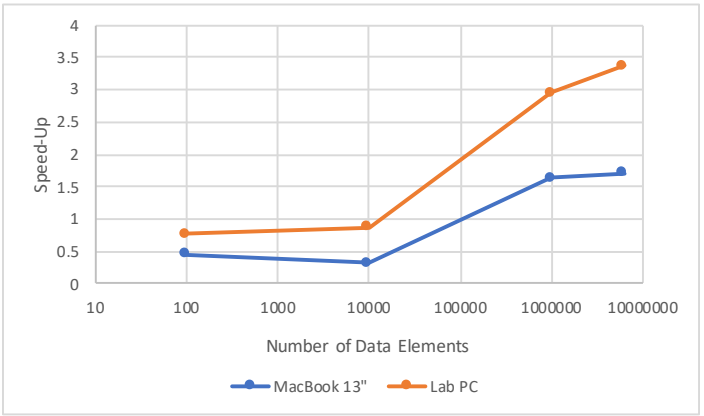


Figure 4: Speedup results as the size of the data increases

Notice that when using less than 10 000 elements, the parallel algorithms (on both architectures) performed *worse* than their serial counterparts. This agrees with the theory discussed previously about the inherent overhead cost of the setting up threads. When handling sets of this size, it would be much more appropriate (and simpler) to use a serial implementation.

As the size of dataset further increased, performance did improve. The Lab PC clearly outperformed the MacBook Pro, but both achieved a speed-up above 1. One must, however, also consider the additional development time involved in creating the parallel implementation, and weigh that up against the marginal speed improvements.

Interestingly, when moving past the 1 000 000 element mark, it seems that the MacBook Pro no longer improves in performance – stagnating at around 1.6x speed-up. The Lab PC, on the other hand, continues to improve, achieving a peak speed-up of just under 3.5x with 10 000 000 elements. This is likely due to the Lab PC having double the number of physical cores as the MacBook, and thus can create and use threads more effectively.

3.2 Dataset Shape

It was worth considering the effect of the 'shape' of the data on the performance results – i.e. the size of the data in each of the three dimensions: t, r, and c.

3.2.1 Number of Time Steps

Firstly, the number of time steps was varied. In order to accurately measure this effect, the total number of data points was kept relatively constant – around 10 000 000 for each iteration. The sequential cutoff was fixed at 1000. Tests were done with parameters:

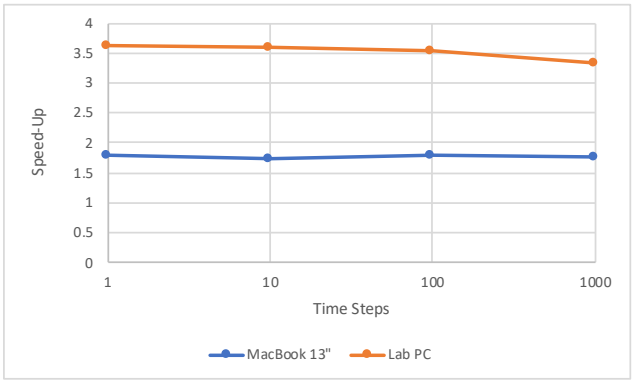
$$(t = 1, r = c = 3160) \quad , \quad (t = 10, r = c = 1000) \quad , \quad (t = 100, r = c = 316) \quad , \quad (t = 1000, r = c = 100)$$

The results of this test are depicted in figure 5a. Notice that, for both architectures, the speed-up changed minimally over the four tests. The Lab PC had much better performance than the MacBook, but that discussion is outside the scope of this test.

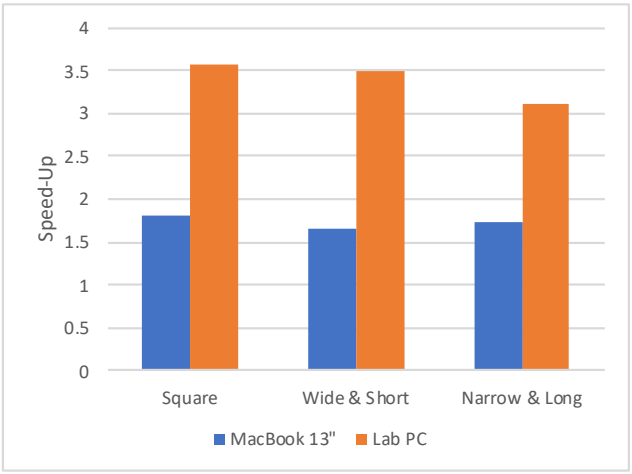
3.2.2 Squareness

Secondly, the so-called 'squareness' of the dataset was considered. The number of time steps was fixed at $t = 50$, and the sequential cutoff was fixed at 10 000. Three cases were then tested, each containing exactly 62 500 elements:
Square ($r = c = 250$), *Wide & Short* ($r = 5, c = 12500$), and *Narrow & Long* ($r = 12500, c = 5$).

The results of this test are shown in figure 5b. Notice again that the speed-up remains relatively constant for each of the three shapes, for each architecture respectively. The *Narrow & Long*'s performance on the Lab PC does deviate slightly from the others, but not significantly.



(a) Varying the number of time steps



(b) Changing the 'squareness' of the dataset

Figure 5: Speed-up as the shape of the dataset changes

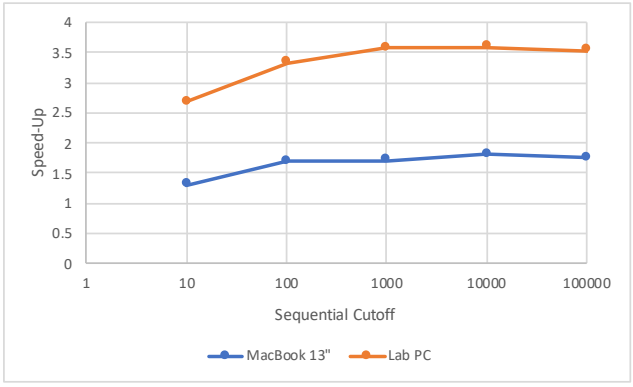
Given these results, one can assume that varying the shape of the data – in any of the three dimensions – should not affect the performance of the parallel algorithm. This indeed follows logically from the process described in section 2.2, as the data is all linearised anyway – regardless of the initial dimensions. Nevertheless, it is good to have confirmed this fact.

This characteristic shows that the parallel implementation is fairly robust, performing well in varying scenarios (provided, of course, there are enough data points – as discussed in section 3.1).

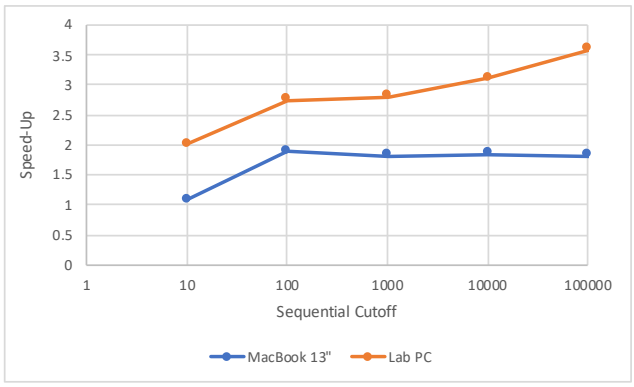
3.3 Sequential Cutoff

3.3.1 The Sweet Spot

Tuning the sequential cutoff value is a key part of achieving the best speed-up possible with a parallel algorithm – as discussed in section 2.5. To test the effect of the sequential cutoff, two scenarios were considered. Firstly, a 'large' dataset with $(t = 100, r = c = 250)$, and secondly a 'medium' dataset with $(t = 25, r = c = 100)$. The sequential cutoff value was varied between 10 and 100 000, in increasing orders of magnitude. Figures 6a and 6b show the results of these two scenarios respectively.



(a) 'Large' dataset scenario



(b) 'Medium' dataset scenario

Figure 6: Speed-up data as the sequential cutoff changes, for two dataset sizes

Performance indeed increased as the sequential cutoff value was enlarged – particularly at the start of the graphs, but the results did not completely agree with what was expected. It was predicted that there would be a so-called 'sweet spot' at which the speed-up would be maximum. Moving above or below this value should have adversely affected performance. However, here it seemed that – particularly in the case of the MacBook Pro – the speed-up performance mostly stagnated for increasing sequential cutoff values. It neither improved nor worsened, and this is odd. An anomaly to this is the Lab PC with the medium dataset (6b), where there was a constant improvement in speed-up, right until the largest speed-up value.

To understand these phenomena better, more investigation would have to be done. More variations of the dataset size are required, and other architectures should be considered. However, the effect likely relates to the relatively large overhead cost of creating a single thread in the algorithm. Creating fewer threads thus ends up being a better suited approach. The number of cores also affects the results – as it determines the degree of parallelism that can be effectively achieved, and hence the occurrence of interesting speed-up when using more cores (the Lab PC) with a smaller dataset.

3.3.2 Ideal Thread Count

One can determine the metric of 'number of threads' using the size of the dataset and the sequential cutoff value. Notice that the full linearised array (section 2.2) has a size of $\dim(t) \cdot \dim(r) \cdot \dim(c) \times 3$. The number of threads is then approximately the size of the array divided by the sequential cutoff value. That is:

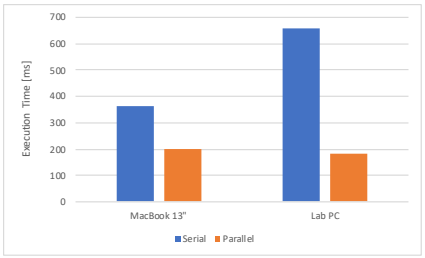
$$\text{n(threads)} \approx \frac{\dim(t) \cdot \dim(r) \cdot \dim(c) \times 3}{\text{Sequential Cutoff}}$$

Using this equation, together with the results in figure 6, it was determined that the optimal number of threads for the MacBook Pro – which had a minor peak at a sequential cutoff of around 100 using the medium dataset – is in the order of magnitude of **7500 threads**. For the Lab PC – which had a maximum speed-up at a cutoff of 100 000 using the medium dataset – its optimal number of threads is in the order of magnitude of **7.5 threads**. This big difference between the two results emphasises the importance of considering the effect of architectures on speed-up.

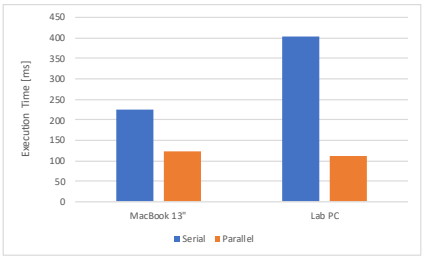
3.4 Time Comparison between Architectures

Up until this point, the results discussed have been focused on the metric of speed-up – how much faster the parallel algorithm runs when compared to the serial algorithm. When comparing the speed-up between the two architectures used, the results have undoubtedly shown that the 2.4GHz quad-core laboratory computer achieves better results than the 2.7GHz dual-core MacBook Pro – figures 4, 5a, 5b and 6 all support this fact. This is logical considering the number of cores present in each architecture – four cores are more useful than two in a parallel problem.

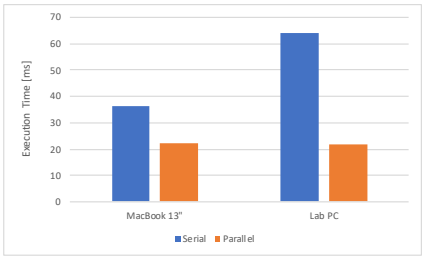
However, it is also interesting to note the actual run-times when comparing the architectures. Consider figure 7.



(a) $t = 1$; $r = c = 3160$; cutoff $= 1000$



(b) $t = 100$; $r = c = 250$; cutoff $=10000$



(c) $t = 1$; $r = c = 1000$; cutoff $= 1000$

Figure 7: Actual run-times achieved for three datasets on both architectures

Notice that in all three tests, the Lab PC has a much slower serial implementation than that of the MacBook Pro – almost half the speed. This can be explained by revisiting the architecture specifications: the MacBook Pro has a 2.7GHz clock, which is faster than the 2.4GHz speed of the Lab PC. Moreover, the MacBook's i5 chip has a 'turboboost' feature, which enables it to increase its clock-speed up to 3.1GHz, provided its power usage and temperature are within an acceptable range. Thus, when running the serial algorithm – which uses a single thread – the MacBook Pro is able to outperform the Lab PC. This advantage, however, is negated when the parallel algorithm is used. The Lab PC can appropriately use all four of its cores, whereas the MacBook only has two cores, and both architectures end up with similar run-times. Nevertheless, the resulting speed-up factor is larger for the Lab PC, due to its slower serial time.

The above discussion highlights the importance of effectively using all cores available in order to actually achieve better performance.

3.5 Optimal Speed-up

Finally, it is useful to note the maximum possible speed-up that was achieved using this parallel algorithm.

For the MacBook Pro: a maximum speed-up of 1.88x was achieved, with the parameters $(t = 25, r = c = 100)$ and a sequential cut-off of 100.

For the Lab PC: a maximum speed-up of 3.61x was achieved, with the parameters $(t = 1, r = c = 3160)$ and a sequential cut-off of 1000.

Theoretically the parallel implementation should be *much* faster (see section 1) – but that 'theory' wildly assumes access to infinite cores with no imperfections whatsoever. This is, of course, unrealistic.

4 Conclusions

The achieved speed-up is by no means phenomenal, and is not even an order of magnitude bigger than the serial implementation. Nevertheless, the algorithm's suitability depends on the context of the problem at hand. Achieving a speed-up of 3x could mean calculating weather predictions within 8 hours instead of 24 hours, and this could prove to be life changing.

It was regularly observed that the MacBook Pro – despite its higher clock-speed (and price) – was less performant than the Lab PC. This is primarily due to the number of cores that each machine has. It is assumed that by running the algorithm on a computer with even more cores – e.g. an octa-core (8) computer – much better results may be found. This notion is supported by the fact that simply moving from two- to four-cores presented a drastic performance improvement.

It is worth mentioning that the reliability of the data presented may be questionable. There are plenty of confounding variables that possibly had an effect on the results – including cache warming, the current load of the computer's CPU, the thermal conditions, and so forth. Much more in-depth and thorough testing is required for more accurate and reliable results. Nevertheless, the findings from this report provide a good starting point, and are appropriate for the scope of this project.

Note that a key component of research was omitted: power consumption. An increasingly important requirement in the modern era is the efficiency of an algorithm's implementation. This simply cannot be overlooked. Moving forwards, all increases in speed should be contrasted with their corresponding increases in power consumption.

More comprehensive research is clearly required for more comprehensive findings.

5 Additional testing: Localised Average Boundary Size

For interest's sake, it is worth considering the effect of expanding the boundary of the localised average calculation when performing classification. In the previous tests, it was declared that the classification at a point depended on the average of the element at that point and its immediate neighbours – i.e. no more than one element away. Now, instead, this localised average was considered to have a boundary of two or three elements in all directions. Pseudocode that enabled using a generic localised average boundary is given in listing 4.

Listing 4: Pseudocode for a generic localised average boundary size

```
static int local = 2; // Localised boundary size
// 'Explode' this data to surrounding elements
for (int r = Math.max(0, i-local); r <= Math.min(data.dimx-1, i+local); r++) {
  for (int c = Math.max(0, j-local); c <= Math.min(data.dimy-1, j+local); c++) {
    data.classification_sum[t*data.dimx*data.dimy + r*data.dimy + c].incr(currX, currY);
  }
}
```

The tests were run on various dataset sizes, with the same sequential cutoff, and the speed-up values were averaged. The results are shown in figure 8.

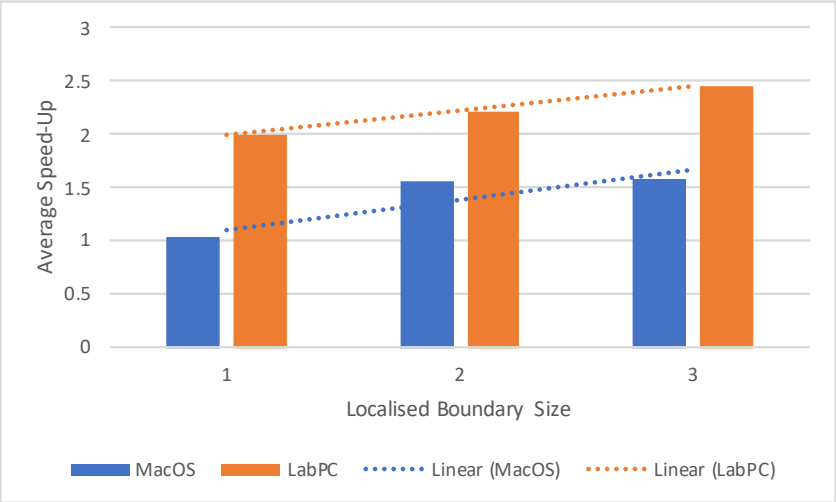


Figure 8: Effect of increasing the localised average boundary size on speed-up

As before, the Lab PC had a larger average speed-up value than the MacBook Pro, but that is irrelevant in this discussion. What matters more is that both architectures had an upward trend in speed-up as the size of the localised average boundary increased, which is interesting. Increasing the localised boundary entails doing more work – the code is required to account for more neighbouring elements at each point. As a consequence, the execution time *increased* – particularly in the serial algorithm. Though the parallel algorithm also suffered, the degree to which is slowed was not as severe. Hence, the speed-up actually *improved* as more work was added to the challenge. At its peak performance, the Lab PC was able to achieve a **4x** speed up with a local average boundary of 3 elements, a large dataset ($t = 1, r = c = 2500$) and a sequential cutoff of 1000 – this was the best speed-up achieved in the entire project.