CSC2002S

Assignment 3

JFFMAH001

Mahmoodah Jaffer
9-13-2019

Contents

Introduction

This assignment involved combining weather simulation with computer programming. Weather simulation when used for prediction purposes is a complex task. These simulations scale in complexity as the size of weather patterns increase. It is very important that these simulations are completed within a certain amount of time in order to be useful.

This assignment focused specifically on cloud classification. For this assignment we had to determine the prevailing wind direction and the types of cloud that might result given a set of data from a weather simulation using both parallel and sequential programming. The prevailing wind over the entire simulation was calculated by finding the average wind vector for all air layers and time steps given to us. The local average wind direction was found and was used to classify the clouds into 3 categories namely cumulus, striated stratus and amorphous stratus.

The main aim of this assignment was to evaluate how parallel and sequential programming perform in terms of speed and efficiency; hence this will be the main topic of discussion in the report below.

Methods

Approach to solution – Sequential

To calculate the prevailing wind of the data given, a method was made that obtained the advection of each element – which contained a vector with x and y co-ordinates. The average for all the x co-ordinate components were obtained and the average for all the y co-ordinate components were obtained. This was then printed to the output text file.

To classify the air layer elements, it was necessary to obtain the local average wind direction (the mean wind of the element and its 8 immediate neighbours). Since not all air layer elements have 8 neighbours (some elements might be on the corner of the matrix) it was necessary to check for edge cases and this can be seen in Figure 1 below which shows the code used to check for edge cases before obtaining the local wind direction.

```
for (int x = -1; x < 2;x++){
   for (int y = -1; y < 2; y++){

      if (j==0 && x ==-1)
         continue;
      if (k==0 && y ==-1)
         continue;
      if (j == (rows-1) && x ==1)
         continue;
      if (k == (columns-1) && y ==1)
         continue;
```

Figure 1: Code that checks for edge cases when getting local average wind direction

Once the local average wind direction of each element was obtained, it was then assigned an integer code based on the type of cloud that was likely to form from that air layer element. This was assigned using the local average wind direction (w) and the uplift value (u).

Approach to solution – Parallelization

The same method was used to find the prevailing wind and to classify each air layer element however since parallel programming was used to do this a different approach was used.

Both programs for the calculation of the prevailing wind and the classification of each element used the Fork/Join framework. Both these programs used the divide-and-conquer algorithm. When creating parallel programs, the sequential cut-off that one sets greatly affects the way that the program will perform. Since the largesample_input.txt file contained 5 242 880 air layer elements the sequential cut-off was set to 4 000 000. For the Fork/Join Framework the number of threads created is determined by the following:

$$Number\ of\ threads\ = \frac{Number\ of\ elements}{Sequential\ Cutoff}$$

There are two types of Fork/Join Task specializations, to calculate the prevailing wind RecursiveTask was used since it returns a value once the computation of the divide-and-algorithm is completed and for classifying the air layer elements RecursiveAction was used since it was not necessary to return a value.

Validation of Algorithm

In order to validate the algorithm, it was necessary to test the output that was obtained by the program against the sample data given to us in the handout for the assignment. This was done by comparing the output for the simplesample_input.txt file given to us with the output that was obtained by the program when using the same file. These can be seen in the figures below:

```
3 2 2
0.333333 0.166667
1 1 1 1
0 0 0 0
0 0 0 0
```

Figure 2: Output given to us for simplesample_input.txt

```
3 2 2
0.33333334 0.16666667
Time taken to find total average = 0.065 seconds
1 1 1 1
0 0 0 0
0 0 0 0
```

Figure 3: Output of program using algorithm

As can be seen from the above figures, the algorithm is correct. The algorithm was tested in the same way using the largesample_input.txt file given to us however the output of this was too large to include in the report.

Timing of algorithms

For correctness purposes, the program was run 5 times and an average was calculated to obtain an accurate time value for each test run. The algorithm was timed using the following methods.

```
static long startTime = 0;

private static void tick(){
   startTime = System.currentTimeMillis();
}
private static float tock(){
   return (System.currentTimeMillis() - startTime) / 1000.0f;
}
```

Figure 4: Code used to obtain run time of program

The tick() method was used to start the system timer and tock() was called after obtaining the values for prevailing wind and classification of wind elements as this would give the time that went by since the tick() method was called.

Measuring Speedup

Speed up was calculated using the following formula

$$Speed\ Up\ = \frac{Sequential\ Time}{Parallel\ Time}$$

Machine Architectures used

Three different machine architectures were used when running both parallel and sequential programs.

Acer Laptop

My personal laptop was used, and the figure below shows the laptops CPU utilization. As can be seen from the figure below the laptop has 2 cores.

Item	Value
OS Name	Microsoft Windows 10 Home Single Language
Version	10.0.18362 Build 18362
Other OS Description	Not Available
OS Manufacturer	Microsoft Corporation
System Name	LAPTOP-FD2ENM43
System Manufacturer	Acer
System Model	Aspire E5-575G
System Type	x64-based PC
System SKU	Aspire E5-575G_115F_1.20
Processor	Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz, 2701 Mhz, 2 Core(s), 4 Logical Processor(s)
BIOS Version/Date	Insyde Corp. V1.20, 13 Dec 2016
SMBIOS Version	3.0
Embedded Controller Version	2.20
BIOS Mode	UEFI
BaseBoard Manufacturer	Acer
BaseBoard Product	Ironman_SK
BaseBoard Version	V1.20
Platform Role	Mobile
Secure Boot State	On
PCR7 Configuration	Elevation Required to View
Windows Directory	C:\WINDOWS
System Directory	C:\WINDOWS\system32
Boot Device	\Device\HarddiskVolume1
Locale	South Africa
Hardware Abstraction Layer	Version = "10.0.18362.356"
User Name	LAPTOP-FD2ENM43\Mahmoodah Jaffer
Time Zone	South Africa Standard Time
Installed Physical Memory (RAM)	8,00 GB

Figure 5: Screenshot of CPU utilization for laptop

Senior Lab i3 Desktop

As the second architecture a PC in the lab was used and the CPU utilization can be seen below. As can be seen from the figure below the number of cores for this architecture is 4.

```
jffmah001@sl-dual-231:~$ lscpu
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
Address sizes:       39 bits physical, 48 bits virtual
CPU(s):              4
On-line CPU(s) list: 0-3
Thread(s) per core:  1
Core(s) per socket:  4
Socket(s):           1
NUMA node(s):        1
Vendor ID:           GenuineIntel
CPU family:          6
Model:               158
Model name:          Intel(R) Core(TM) i3-8100 CPU @ 3.60GHz
Stepping:            11
CPU MHz:             800.086
CPU max MHz:         3600.0000
CPU min MHz:         800.0000
BogoMIPS:            7200.00
Virtualization:      VT-x
L1d cache:           32K
L1i cache:           32K
L2 cache:            256K
L3 cache:            6144K
NUMA node0 CPU(s):   0-3
Flags:               fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe s
yscall nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf tsc_known_freq pni pcl
mulqdq dtes64 monitor ds_cpl vmx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx
 f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault invpcid_single pti ssbd ibrs ibpb stibp tpr_shadow vnmi flexpriority ept vpid ept_ad fsgsba
se tsc_adjust bmi1 avx2 smep bmi2 erms invpcid mpx rdseed adx smap clflushopt intel_pt xsaveopt xsavec xgetbv1 xsaves dtherm arat pln pts hwp
hwp_notify hwp_act_window hwp_epp md_clear flush_l1d
jffmah001@sl-dual-231:~$
```

Figure 6: Screenshot of CPU utilization for lab PC

MacBook

For extra credit a third architecture was used to test the sequential and parallel programs. The device used was a MacBook and its CPU utilization can be seen below. The MacBook has 2 cores.

Figure 7: Screenshot of CPU utilization for MacBook

Interesting Observations

When first writing the code for this assignment I ran it on my virtual box and there was very little speed up – this was due to the fact that since I was running the programs on a virtual box and not on an actual machine the virtual box only had a certain amount of space that it could use. When a ran it on the lab PCs the speed up was a lot greater.

Results and Discussion

Effect of the number of threads

In order to demonstrate the effect of the number of threads on parallel speed up it was necessary to use one data size (the largesample_input.txt file given to us) and then change the sequential cut-off of the programme and hence change the number of threads run by the parallel program.

Table 1 below shows the different sequential cut-offs chosen for the parallel program and hence the number of threads. The number of elements in the data file is 20 X 512 X 512 = 5 242 880.

Sequential Cut-off	Number of threads
500 000	10
1 000 000	5
2 000 000	2
2 500 000	2
4 000 000	1
5 000 000	1

Table 1: Different sequential cut-offs for parallel program

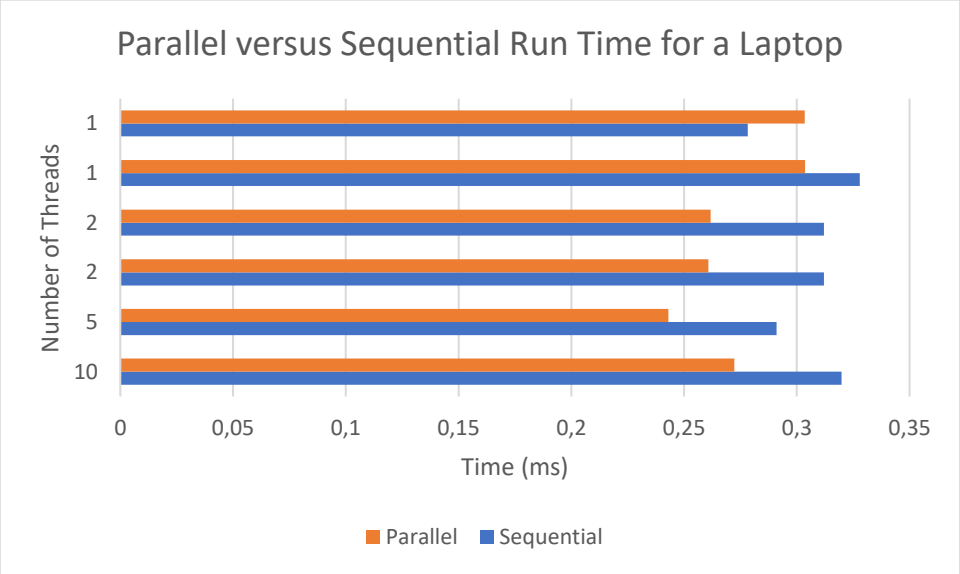

Figure 8: Graph showing the run time of the sequential and parallel programme when being run on a laptop for a different number of threads

As can be seen from the Figure above, the parallel program runs faster than the sequential program for the most part. When the sequential cut-off was set to 5 000 000 the sequential program was faster than the parallel program however this is wrong and could be due to programs running in the background that took up space on the CPU at the time that it was running. The optimal number of threads when running the programs on the laptop was 2 as this was when the parallel program ran a lot faster than the sequential. This is as expected since the laptop is dual-core.

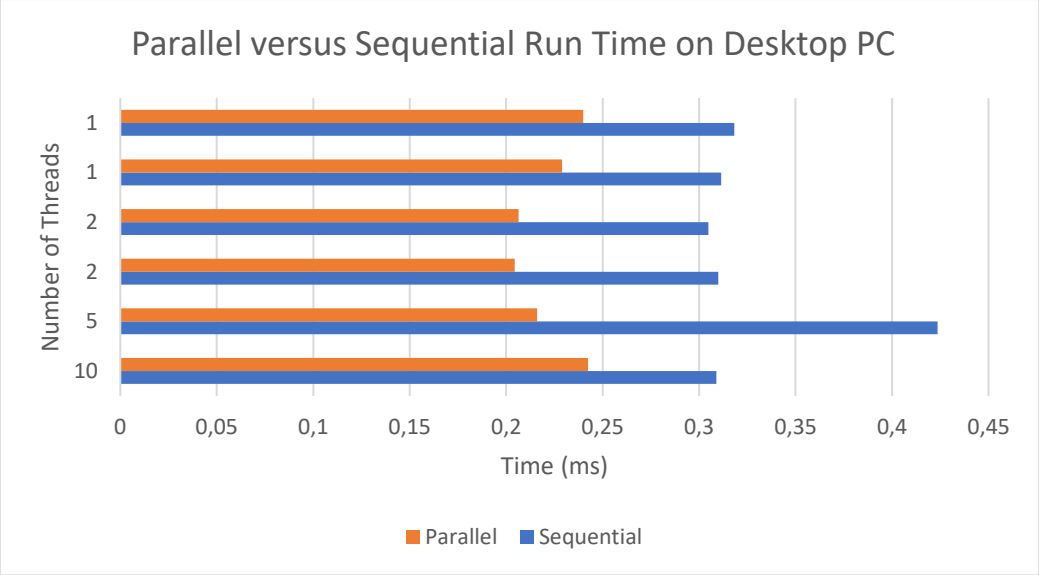

Figure 9: Graph showing the run time of the sequential and parallel programme when being run on a desktop PC for a different number of threads

From the Figure above, we can see that the parallel program is always faster than the sequential program. It seems like 5 threads was the optimal amount of threads when being run of the desktop PC in the labs. This is as expected since the PCs in the lab have a 4 core CPU.

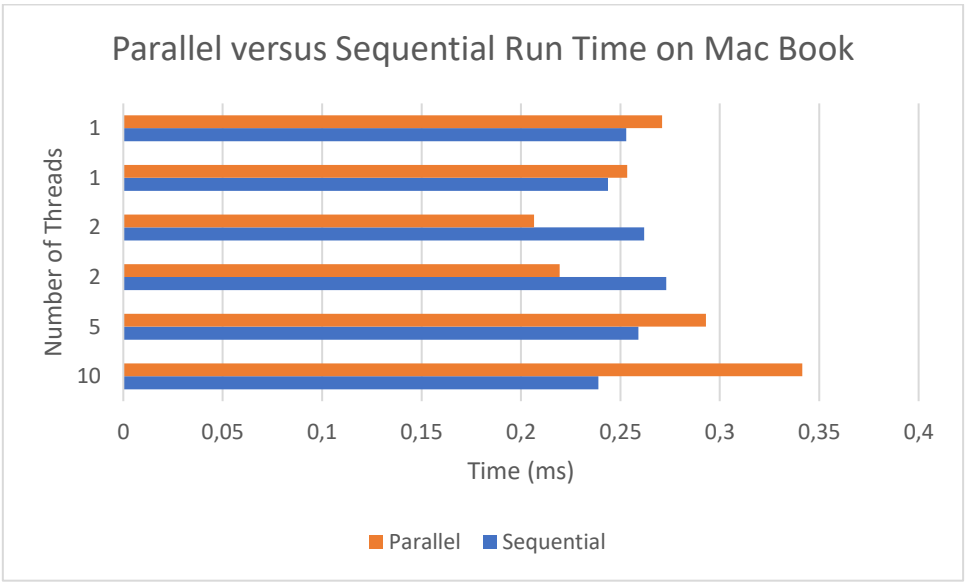

Figure 10: Graph showing the run time of the sequential and parallel programme when being run on a MacBook for a different number of threads

As can be seen from the above Figure, the sequential program runs faster than the parallel program most of time – except when the number of threads is two then the parallel program runs faster. Therefore, the optimal number of threads on this architecture is two. This is not the optimal performance of parallelization.

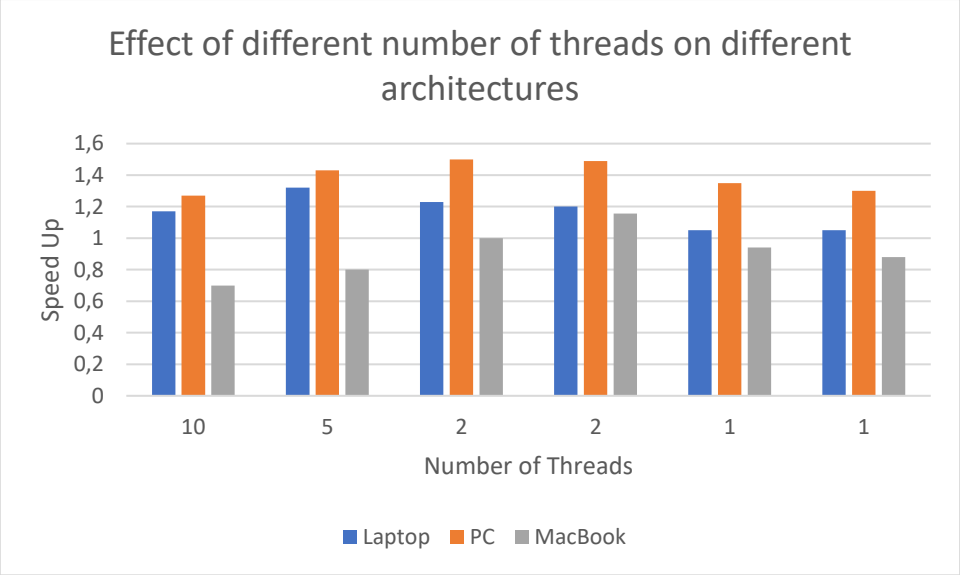

Figure 11: Graph showing the effect that changing the number of threads has on the speed up of parallel programs for different architectures

From the figure above, it can be seen that overall the PC from the lab had the best speed up and this is to be expected as it has 4 cores and both the laptop and MacBook only have 2 cores. Overall, the best number of threads seems to be 2 as this produced higher speed ups when comparing all architectures. Hence, the optimal sequential cut-off for this problem is approximately 2500 000.

Effect of data sizes

In order to demonstrate the effect of data sizes on parallel speed up it was necessary to use the same sequential cut-off and then change the data sizes of the text files. The sequential cut-off for all data sizes was set to 4 000 000. The following data sizes were used:

Data Size	Number of Elements
20 X 500 X 500	5 000 000
20 X 512 X 512	5 242 880
20 X 600 X 600	7 200 000
20 X 700 X 700	9 800 000

Table 2: Different data sizes used for sequential and parallel program

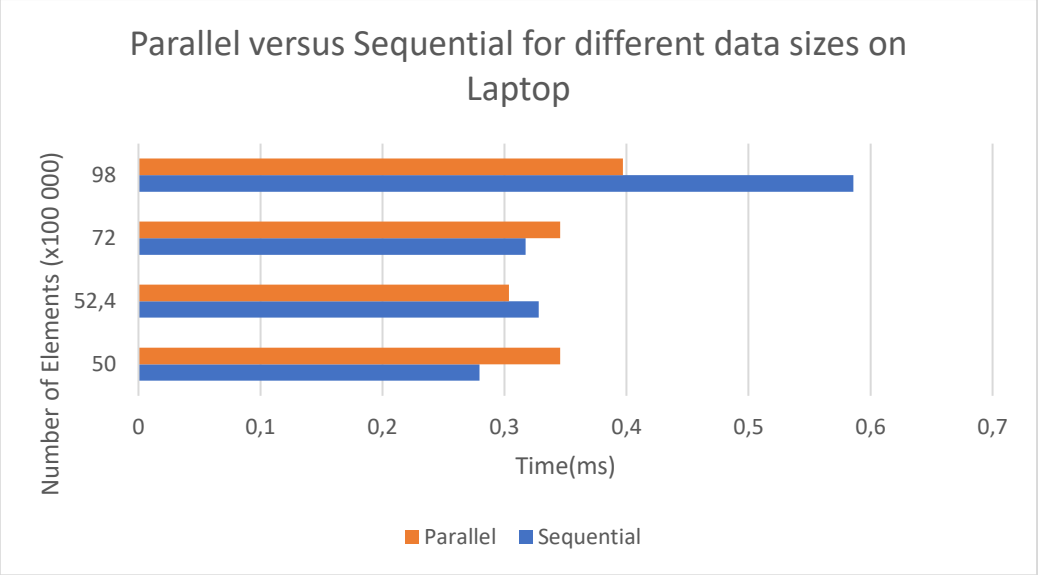

Figure 12: Graph showing the run time of the sequential and parallel programme when being run on a laptop for different data sizes

As can be seen in the figure above, the parallel program did not perform as expected as it took longer to run than the sequential program did most of the time. The only exception to this was when the number of elements was increased to 9.8M. A possible reason for this could be that other programs were running in the background while the programs were running. Although the parallel program did not behave as expected in terms of running faster than the sequential program, the run time of the parallel program remained relatively similar whereas with the sequential program as the data size increased the run time for the sequential program increased as well.

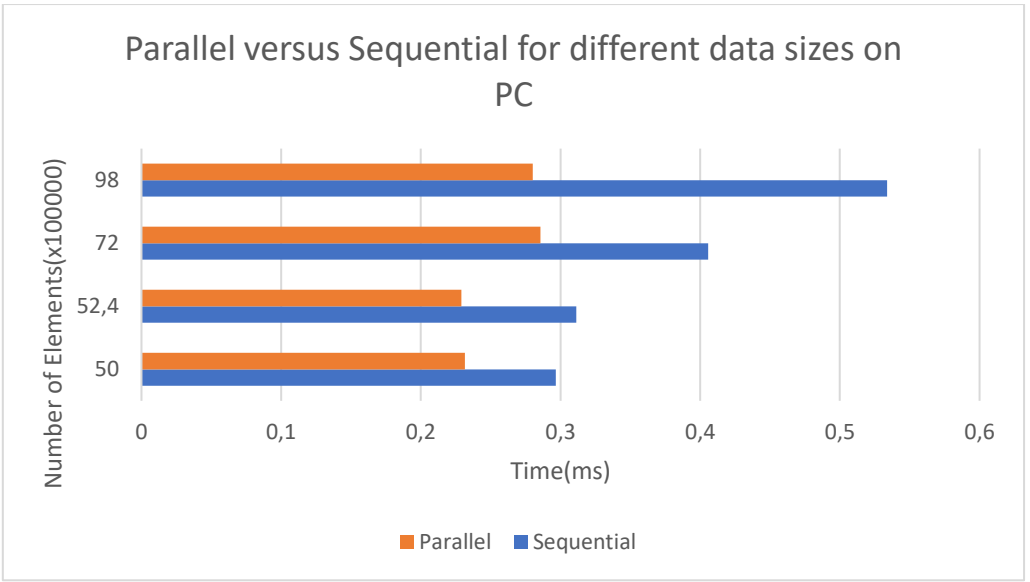

Figure 13: Graph showing the run time of the sequential and parallel programme when being run on a desktop PC for different data sizes

From the figure above, it can be seen that the parallel program performed very well on the PC as it performs as expected which is that it runs faster than the sequential program. We can see that for the sequential program as the data size increases the run time of the sequential program increases.

However, for the parallel program as the data size increases the run time of the parallel program remains relatively the same.

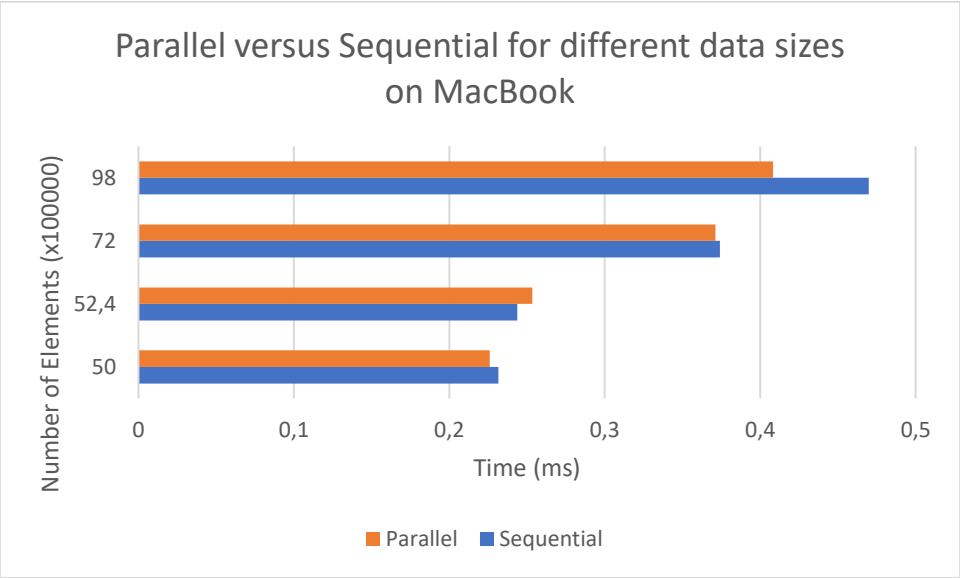

Figure 14: Graph showing the run time of the sequential and parallel programme when being run on a MacBook for different data sizes

From the above figure, we can see that the parallel program does run faster than the sequential program, however as the data size increases it seems like the run time of both the parallel and sequential program increases.

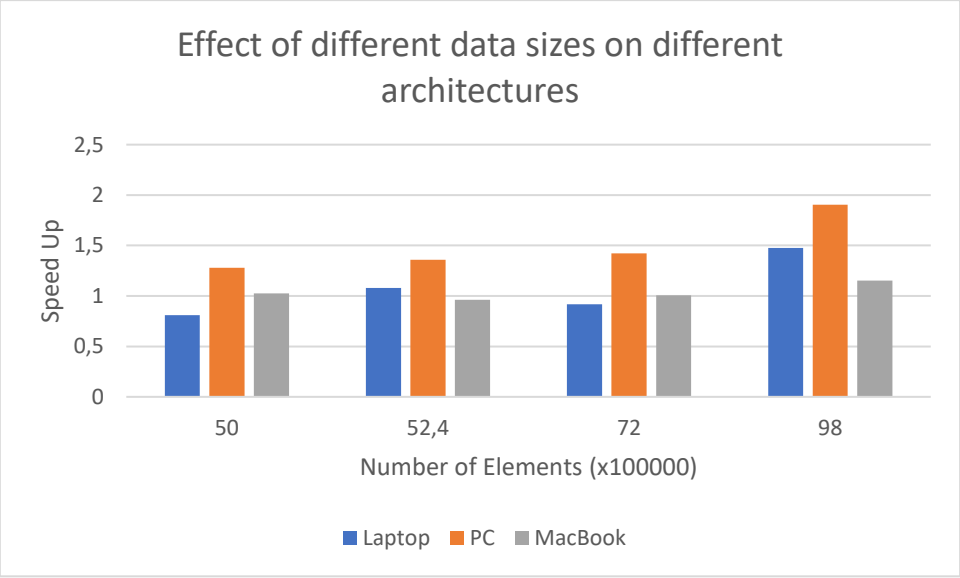

Figure 15: Graph showing the effect that changing data sizes has on the speed up of parallel programs for different architectures

Once again, as seen previously when discussing the effect of a different number of threads on parallel speedup the lab PC performed the best and this is as expected since it has 4 cores. This time however, it seems like the MacBook performed better than the laptop in terms of speed up. A speedup of more than x1 was achieved for all three architecture but it seems like the parallel

program performs well for data sizes that are have 9.8M elements or more since this is when the parallel program has the best speedup for all 3 architectures.

From the above discussions it can be said that it is worth using parallelization to tackle this problem in Java since the parallel program runs a lot faster than the sequential program and hence saves time especially when the size of data sets increases in size. The parallel program is more efficient than the sequential program and therefore it should be used to tackle this problem instead of the sequential program.

To calculate the ideal speed up Amdahl's Law must be used:

$\text{Amdahl's Law} = \frac{1}{1 - p + \frac{p}{n}}$, where p = 0.998 and n is the number of cores.

Therefore, for the 2-core laptop and MacBook the ideal speed up is 1.996 which is approximately a speed up of 2. For the 4-core lab PC the ideal speed up is 3.97. From the above results, we can see that we only get half of the ideal speed up when running the parallel program and this is probably due to the fact that the CPU is running other programs at the same time that the parallel program is being run.

Conclusions

In conclusion for the most part the results seem to be as expected however there were some results that did not perform as expected. This was probably due to the fact that there were other factors that affected the performance of the parallel program such as running other programs while running the parallel program as this could have caused the parallel program to run slower.

I discovered that in order to create an efficient parallel program, it is necessary to break the problem up into smaller pieces of work that can be solved simultaneously within a parallel program. It was found that the more cores a device has the better the performance of the parallel program will be.

The biggest takeaway from this assignment was that it is necessary to use parallel programming to solve large problems that would normally take a long period of time to complete if parallel programming was not used as it is more efficient, and it saves time.