Parallelism Assignment

Ryan Anderson (ANDRYA005)

31 August 2019

Contents

Introduction

The aim of this project is to (1) compare the run times of serial and parallel programming across a range of input sizes as well as (2) experiment within the parallel program in order to determine the limits at which sequential processing should begin. We are using the Java *Fork/Join* framework in the context of cloud classification and prevailing wind calculations.

Two divide and conquer algorithms are being deployed in order to implement parallelism. The *Map Pattern* was used when performing cloud classification, whereas the *Reduce Pattern* was used to calculate the prevailing wind over all the wind vectors.

The following theoretical run times are expected:

$$T_1 = O(n)$$
$$T_\infty = O(log(n))$$

where T_1 represents the work of the program and T_∞ represents the span. Thus, the theoretical speedup from using parallelism should be $\frac{n}{log(n)}$. However, the span assumes that the program has access to an infinite number of processors, thus we would expect a far smaller theoretical speedup.

Methods

We will now give a description of our approach to the solution, with details on the parallelization.

As mentioned in the introduction, a combination of the *Map Pattern* and *Reduce Pattern* were used. The *Map Pattern* was applied to assign the type of cloud that is likely to form in that location based on a comparison of the local average wind direction and uplift value. This was suitable because we were able to operate on each location independently without having to combine results. The *Reduce Pattern* was used to calculate the prevailing wind over all of the locations and time steps. This pattern was suitable because we needed to produce a single answer (the sum of all x advection values and y advection values) from an array via addition (an associative operator).

The parallelization of the prevailing wind and cloud classification was done in one class, `SumLocal`. The class extended `RecursiveTask` as we needed to return an array containing the sum of all of the x-advection and y-advection values. Inside the *compute* method, we utilized a divide and conquer algorithm in order to improve performance, applying parallelism to the recursive calls thus, spreading the accumulation of partial results across the threads. Each time the *compute* method was called, we created two `SumLocal` objects, `left` and `right`, dividing the data in two - resulting in a binary split. The left object was then *forked* and the right object *computed*. This halves the number of threads required and thus reduces the overheads associated with instantiating these threads, without compromising on speed.

In theory, dividing down to a single element gives optimal speedup, however, the overheads associated with creating all these threads generally outweighs the increased performance. Thus, we introduced a sequential cutoff into the *compute* method, after which the cloud classification and prevailing wind calculations were done sequentially.

To determine the optimal sequential cutoff, we did the following for each input size and architecture:

1. Looped through the array, {400,800,1200,1600,2000,2400,2800,3200,3600,4000}, of sequential cutoffs. After a few experiments, we felt like this range was suitable for the problem.
2. For each sequential cutoff, we ran the parallel program. We called `System.currentTimeMillis()` before and immediately after invoking the forkjoin pool.
3. The difference between the two times was then calculated and used as the run time.
4. *2* and *3* were repeated 15 times, and the corresponding 15 results were written to a text file.

Thus, each run of the parallel program created 10 text files, one for each sequential cutoff value.

The above process was repeated with different sample input files. The size of these files were in the range {500, 50 000, 200 000, 1 800 000, 20 000 000}. Again, these input sizes were deemed to be an appropriate spread, after some experimentation. These sample input files were created using a random number generator, sampling from the open interval (0,1). Furthermore, each input size and sequential cutoff was tested on two different architectures, a 2 core Windows laptop, and a 4 core Ubuntu desktop.

Once all of the text files containing the parallel run times for each architecture, input size and sequential cutoff had been produced, we then read these files into data frames using *R* and calculated the average run times of the 15 recorded run times for each architecture, input size and sequential cutoff.

In order to calculate the speedup of the parallel program for each architecture and input size, we used the formula:

$$speedup_{ijk} = \frac{ave_{ij}^{Sequential}}{ave_{ijk}^{Parallel}}$$

where $ave_{ij}^{Sequential}$ denotes the average run time of our sequential program with architecture i and input size j and $ave_{ijk}^{Parallel}$ denotes the average run time of our parallel program with architecture i, input size j and sequential cutoff k. Reiterating what was mentioned above, $i \in \{Ubuntu, Windows\}$, $j \in \{500, 50000, 200000, 1800000, 20000000\}$ and $k \in \{400, 800, 1200, 1600, 2000, 2400, 2800, 3200, 3600, 4000\}$.

The outlined process above was for how $ave_{ijk}^{Parallel}$ was calculated, now we will outline how $ave_{ij}^{Sequential}$ was calculated.

For each architecture and input size, we did the following:

1. Ran the sequential program calling `System.currentTimeMillis()` before and immediately after invoking two methods: `prevailingWind()` - to calculate the prevailing wind - and `classifyAll()` - to assign the cloud type to the location.
2. The difference between the two times was then calculated and used as the run time.
3. *1* and *2* were repeated 15 times, and the corresponding 15 results were written to a text file.

Once all of the text files containing the sequential run times for each architecture and input size had been produced, we then read these files into data frames using *R* and calculated the average run times of the 15 recorded run times for each architecture and input size.

To calculate the optimal number of threads[1] for each architecture and input size, we followed the same method as outlined for calculating the optimal sequential cutoff value, however, we added a counter into the *compute* method which incremented each time the method was called. This counter represented the number of threads that had been created. The total for the counter was then written to a text file in the same way that the parallel run time was for each cutoff. Thus, we had a text file for each architecture, input size and sequential cutoff containing 15 total counter values. We then read these files into data frames using *R* and calculated the average number of threads created from the 15 recorded total counter values for each architecture, input size and sequential cutoff.

We then plotted the average parallel run times, the average speed up and the average number of threads created in order to answer the questions of interest for the assignment.

We validated the parallel algorithm by creating a java file called `CloudClassification.java` which took two arguments, *file1* and *file2* - where *file1* was the output file created by the parallel program, and *file2* was the provided sample output file. The program then used two scanners (one for each file) and the `.equals()` method to check for equality in each line of the two files. If a difference was detected, the user was notified that the output file produced by the parallel program was incorrect.

Finally, I encountered a few interesting problems:

[1] "number of threads" refers to the number of binary splits performed. Because we only created threads on half of these binary splits, the actual number of threads is half this value.

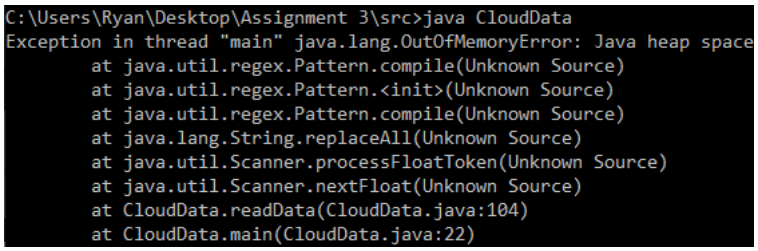

Figure 1: Error Message

1. The Windows laptop did not have enough Memory to read in 20 million data points, whereas, the Ubuntu Desktop could. Thus, the sample input size of 20 million was only applied to Ubuntu. *Figure 1* shows the error message.
2. For the sequential and parallel programs, the first execution of the programs had a far higher run time than subsequent executions. James Gain mentioned in lectures that this was due to Java caching certain values after the first run. Thus, this value was ignored when calculating averages.
3. When looping through the 15 runs, there seemed to be a pattern where approximately every 5 runs, the run time would be significantly higher than the majority of the run times. This occurred for both the sequential and parallel programs. We do not have an explanation for this.
4. For a given sequential cutoff, architecture and input size, the number of threads used for each run was sometimes marginally different. This is an excellent example of a race condition in the code. In theory, the number of threads should be constant, but this race condition is causing non-determinism. In order to negate the differences between the number of threads in each run, we are using the average number of threads over 15 runs. It is noted that we should use locks in order to fix this issue, however, this was deemed out of the scope of the assignment.

Results and Discussions

We will start off with a discussion of the Ubuntu results, followed by a discussion of the Windows results. For each architecture we will discuss the following:

1. For what range of data set sizes does your parallel program perform well?
2. What is the maximum speedup obtainable with your parallel approach?
3. What is an optimal sequential cutoff for this problem?
4. What is the optimal number of threads on each architecture?

On each run time graph, the dotted red line represents the average sequential performance for the input size of interest. On the average speedup graphs, the dotted red line at *Average Speedup*=1 (no speedup) is used as a benchmark to compare how good/bad the speedup was.

UNIX RESULTS

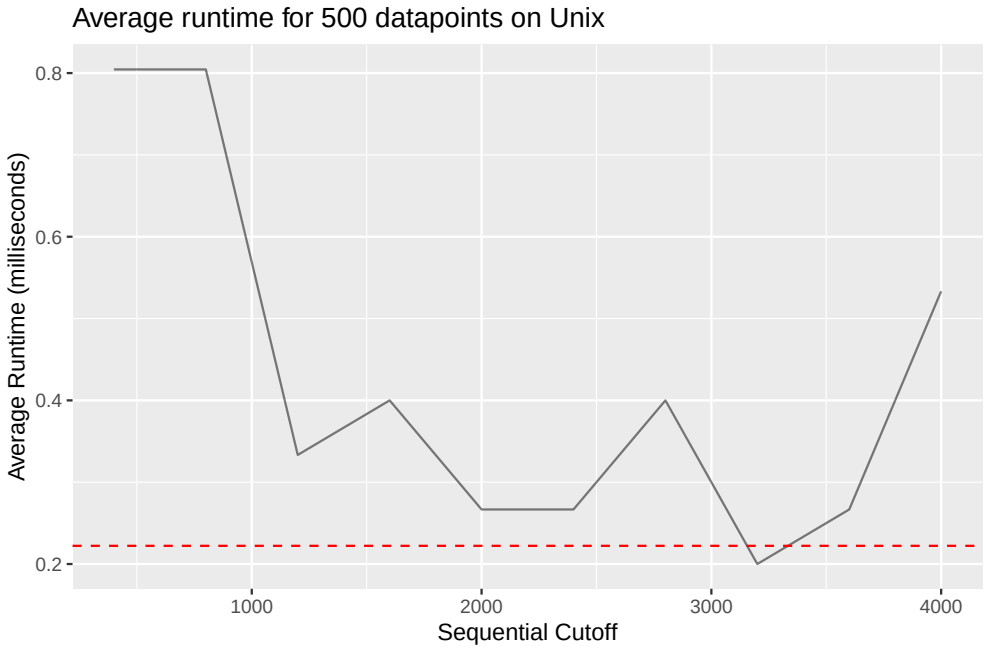

Figure 1

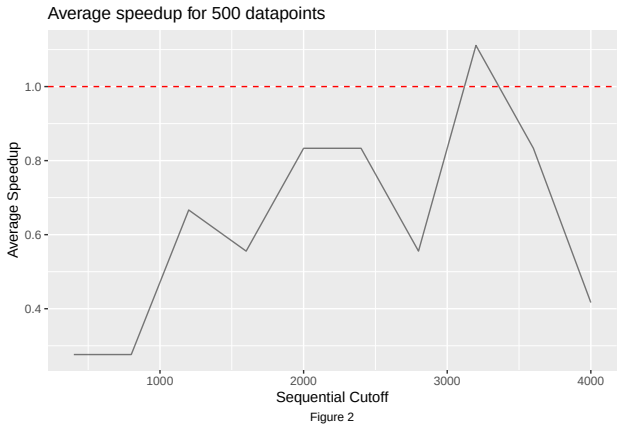

Figure 2

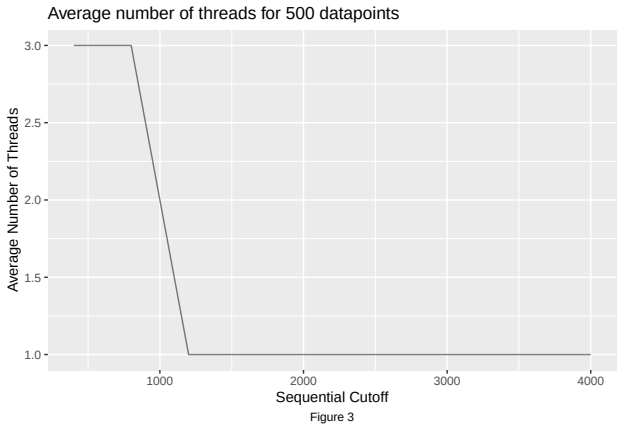

Figure 3

From *Figure 1* above, it is clear that, for 500 data points, the parallel program only outperforms the sequential program at a sequential cutoff of 3200. As can be seen in *Figure 3*, this corresponds to one thread being used, as the data size is less than the sequential cutoff. We would expect to see marginally higher run times in the parallel program compared to the sequential program for all sequential cutoff values greater than 1200 (where the number of threads used is one) because, in these cases, the parallel program is effectively running sequentially, however, the overhead of creating the one thread increases the run time marginally.

For this input size, it is clearly not worth using threads as the overheads associated with creating the threads swamps the savings of implementing them, increasing the run time. This is evident in *Figure 2*, where the parallel program (using 3 threads) is approximately 10 times slower (0.1 times faster) than the sequential program

Thus, for this small input size, it is better to run the program sequentially than in parallel.

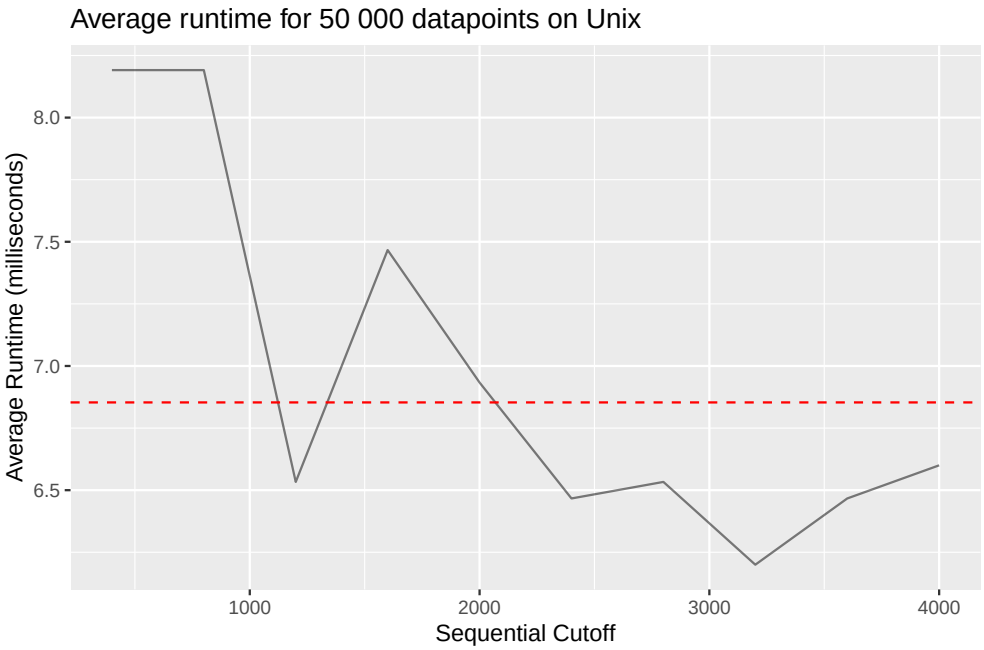

Figure 4

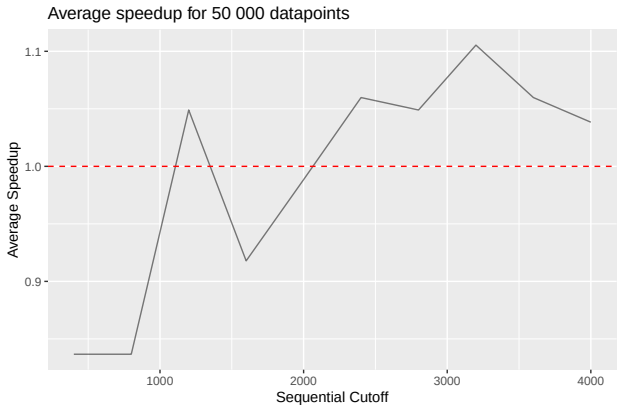

Figure 5

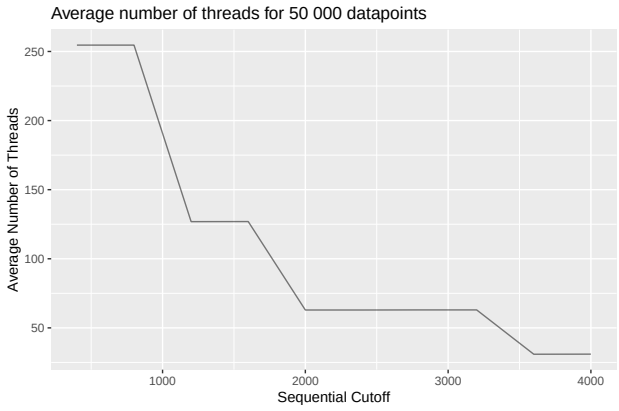

Figure 6

As is evident in *Figure 4* and *Figure 5* above, for 50 000 data points, the parallel program outperforms the sequential program for all sequential cutoffs greater than 1200 (bar 1600). In general, it appears that too much threading (i.e. lower-valued sequential cutoffs) is worsening the run time of the parallel program, suggesting again that the overheads of creating these threads are swamping the speedup of using them.

The optimal sequential cutoff seems to be around 3200 (*Figure 4*), where 26 threads were used (*Figure 6*) - running approximately 1.1 times faster than the sequential program (*Figure 5*).

However, it does not seem necessary to use a parallel program for 50 000 data points as a speed up of 1.1 (approximate run time of 6.25ms for parallel and 6.87ms for sequential) is not very significant. Thus, it would be more appropriate to use the sequential version because it is simpler to write, understand and only takes approximately 0.6ms longer to run.

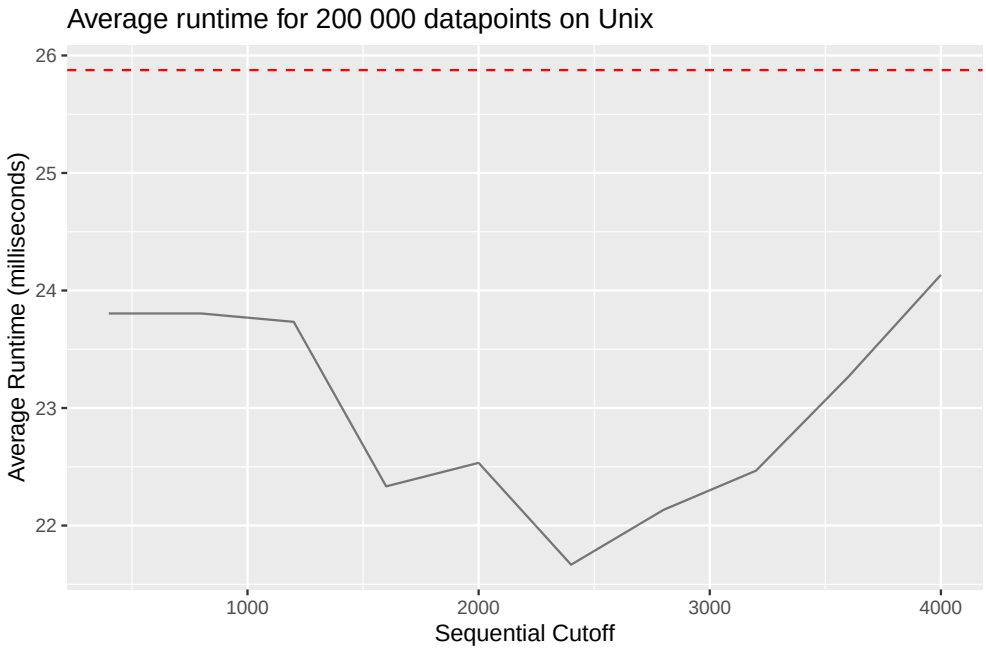

Figure 7

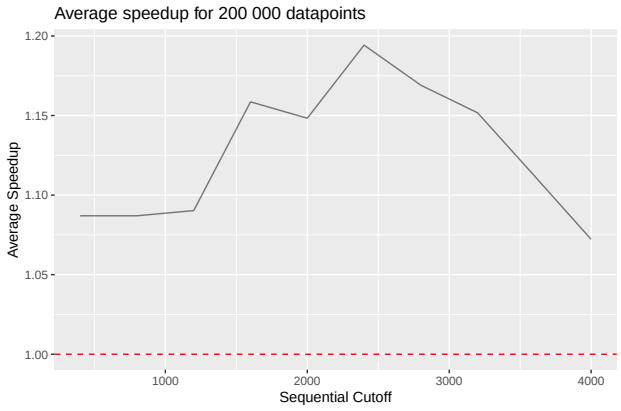

Figure 8

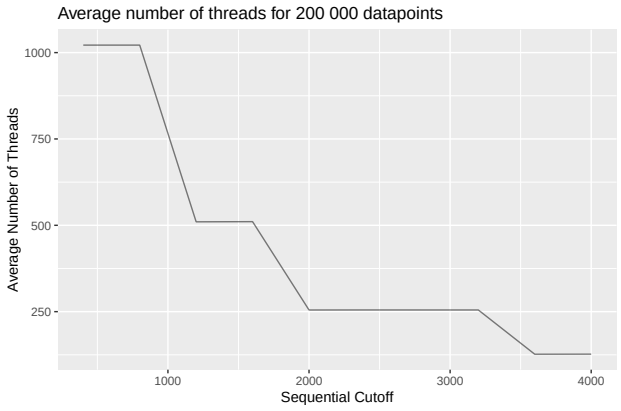

Figure 9

For 200 000 data points it is clear that the benefits of parallelism are really coming into effect. From *Figure 7* and *Figure 8*, it is clear that for all sequential cutoff values, the run time for the parallel program is lower than the sequential program - and it follows that the speedup is greater than 1.

The optimal sequential cutoff appears to be at 2400. At this value, the program used 250 threads with a speedup of approximately 1.2x.

We seem to be seeing less noise in the results as *Figure 7* and *Figure 8* are taking on the parabolic shape that you would expect to see. Thus, it seems that we have reached the input size needed to really consider the parallel program's results seriously.

As previously mentioned, the optimal speedup is only 1.2x, however, we do not expect to see a huge speedup in this assignment because the work performed sequentially is not very computationally expensive and thus the effects of parallelism are not as large as in other contexts.

But, it still seems like using a parallel program is not necessary for this input size. We only achieved an approximate 5ms saving (1.2x speedup) which really is negligible. Thus, it would be better to perform the task sequentially.

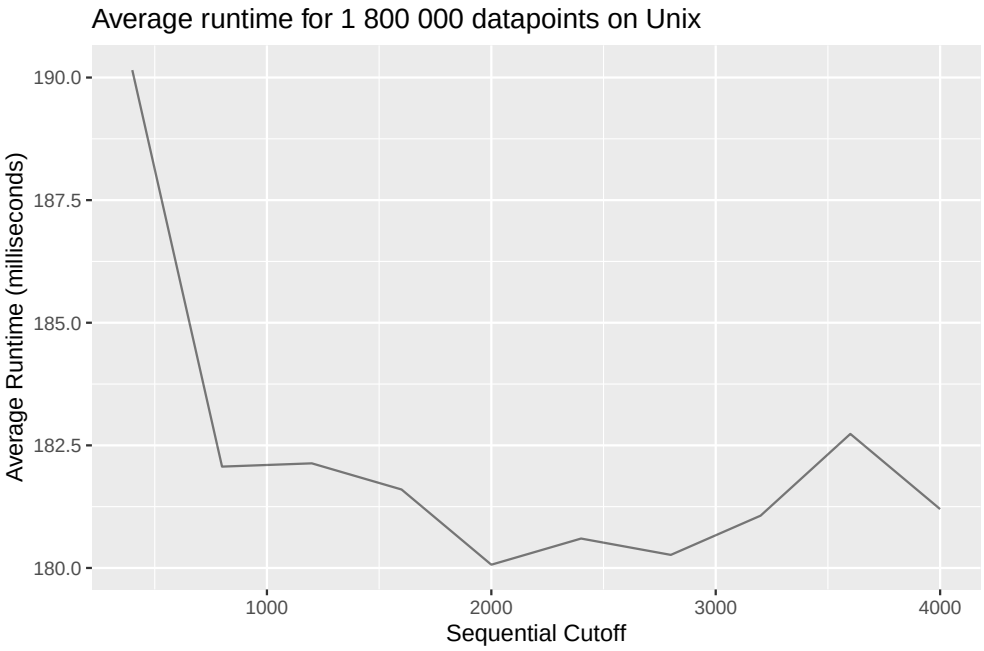

Figure 10

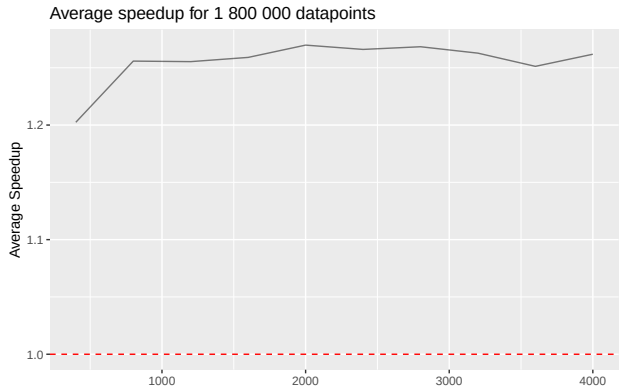

Figure 11

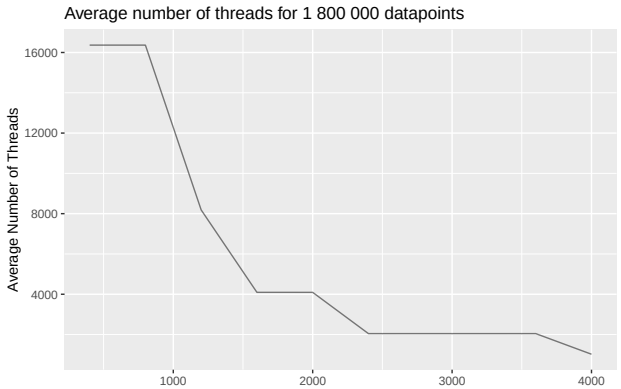

Figure 12

For 1 800 000 data points the results are slightly more in favour of parallelism.

The sequential average was excluded from *Figure 10* as it distorted the scale too severely, but it was 228.64. Thus, you can see in *Figure 10* that we are now getting a larger saving in time, however, from *Figure 11* we can see that the speedup is still not flattering - still a very modest 1.3x at most.

From *Figure 10* we can see that the optimal sequential cutoff is 2000, however, it is only marginally better than a sequential cutoff of 2800. With a sequential cutoff of 2000, we achieved a speedup of approximately 1.28x, which was better than what was seen in the other input sizes. However, this is still not very significant. This is emphasized by the fact that we are only saving approximately 48ms. This is particularly disappointing when considering that 1 800 000 data points would be expected to be large enough to see a significant improvement.

Thus, it again seems like a sequential execution would be preferred.

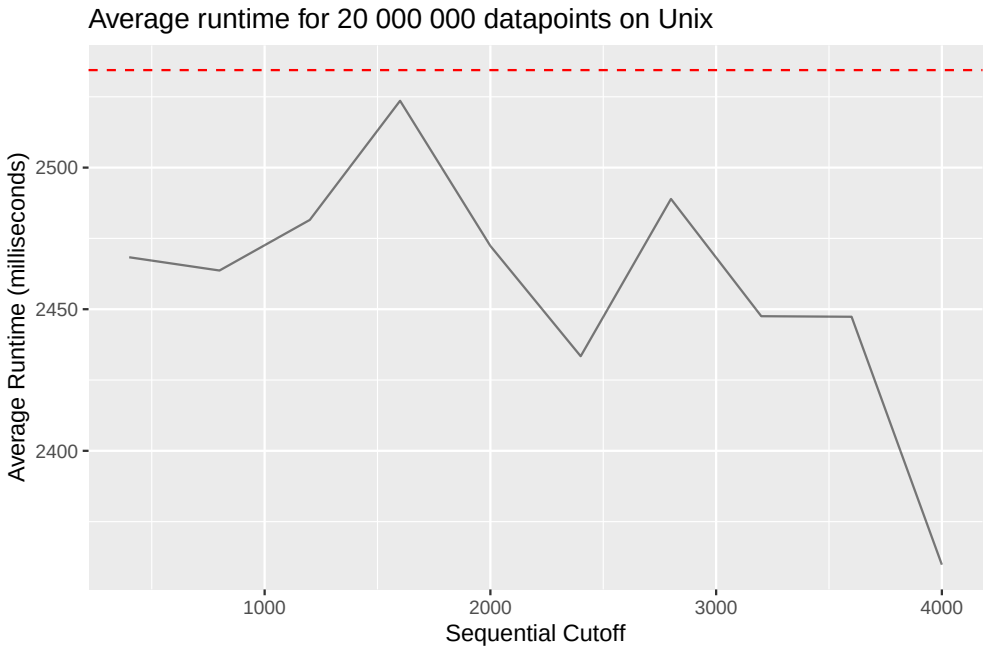

Figure 13

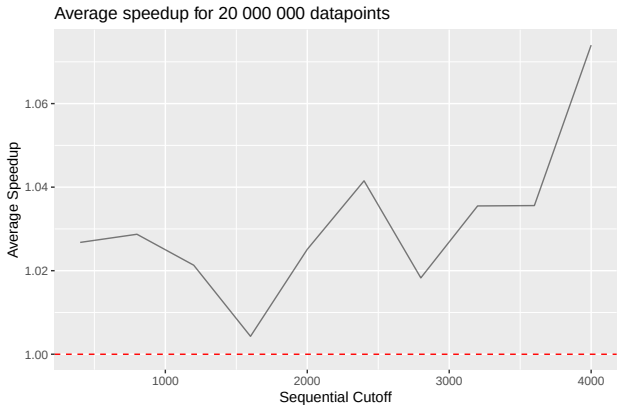

Figure 14

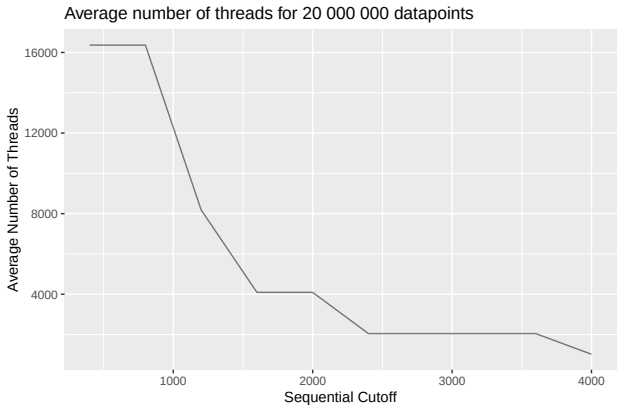

Figure 15

Finally, we performed the same tests on 20 000 000 data points. This produced some rather interesting results.

From *Figure 14*, it appears that the speedup has decreased in comparison to the 1 800 000 data points case, over the range of sequential cutoffs. This seems rather counter-intuitive, as you would expect a parallel program to perform far better than the sequential program when the data set is this large, due to the number of sequential operations that have to performed. As can be seen by *Figure 13*, the reason for this could be that the optimal sequential cutoff is higher than 4000 and thus is not represented over our range.

It is also noteworthy that the sequential program almost performed equivalently to the parallel program when a sequential cutoff of 1600 was used. A possible explanation for this is that 16 000 threads (*Figure 15*) were used for that example, causing a large amount of threading overheads, slowing down the program severely.

It is inconclusive whether the parallel or sequential program should have been used here due to the restriction of the range of sequential cutoffs. However, over our range of sequential cutoffs, the sequential program should have been used, due to the very minimal speedup obtained.

On a whole, it seems that the benefits of parallelism increased as the size of the data increased (bar the 20 000 000 case - however, these results were inconclusive). It was concluded that the sequential version of the program should have been used for all the input sizes because the speedups obtained were minimal and due to the extra simplicity of reading and writing sequential code. However, it was noted that, given a different task to fulfill, particularly one that involved more sequential work, the speedup would probably have been more significant.

We will now present the results from the Windows machine.

WINDOWS RESULTS

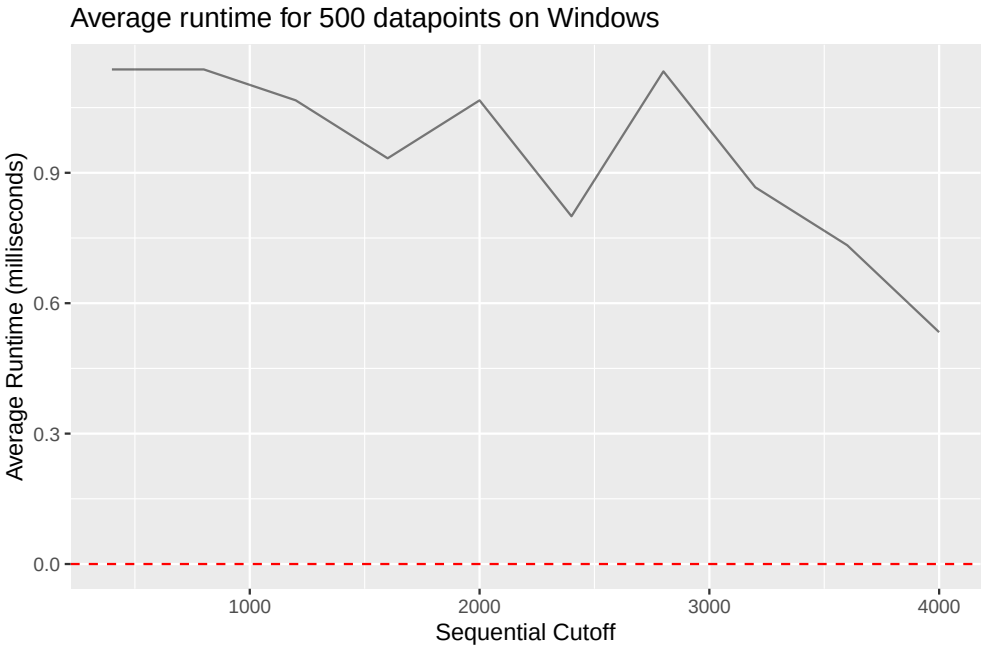

Figure 16

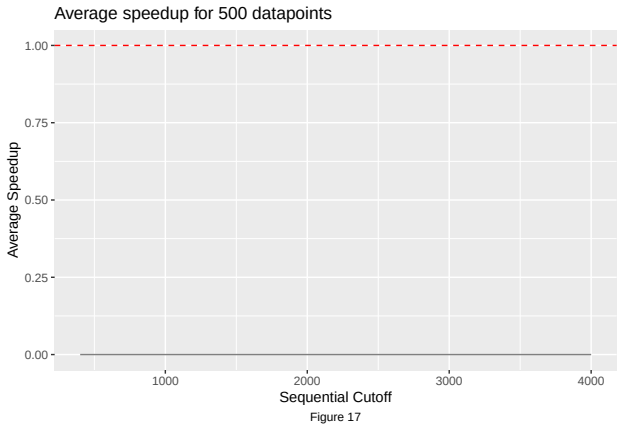

Figure 17

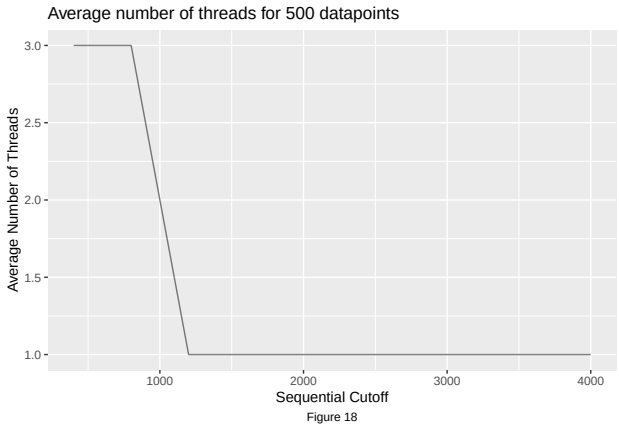

Figure 18

Considering 500 data points, from *Figure 16* and *Figure 17* it is clear that the sequential program outperformed the parallel. The sequential program averaged a run time of approximately 0.02ms which seems plausible due to the size of the input data.

Figure 17 does not provide much useful information because the numerator for the speedup calculation is zero, resulting in the speedup being zero.

The optimal sequential cutoff seems to be greater than or equal to 4000 due to the trend of the graph. It makes sense that optimal sequential cutoff corresponds to a program that uses only one thread, because the overheads of creating numerous threads for such a small input data size offsets the benefits.

Furthermore, as in the Unix results, it seems that the difference between the sequential results and the parallel results when one thread was used, is due to the overheads of instantiating and implementing the one thread.

As the average run time for the sequential program is approximately 0.02ms, the sequential program should definitely be used.

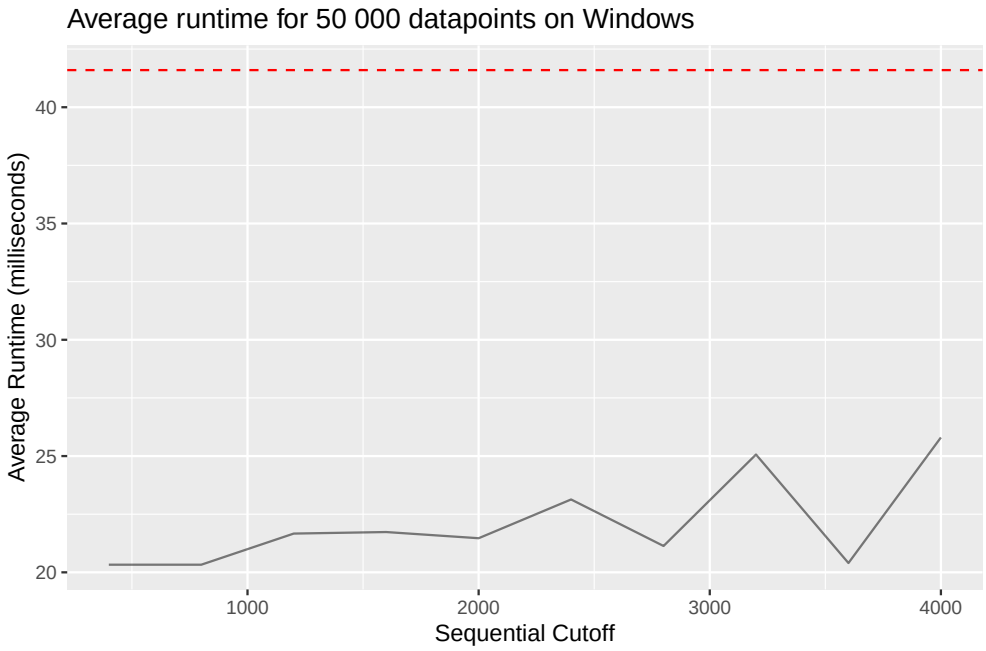

Figure 19

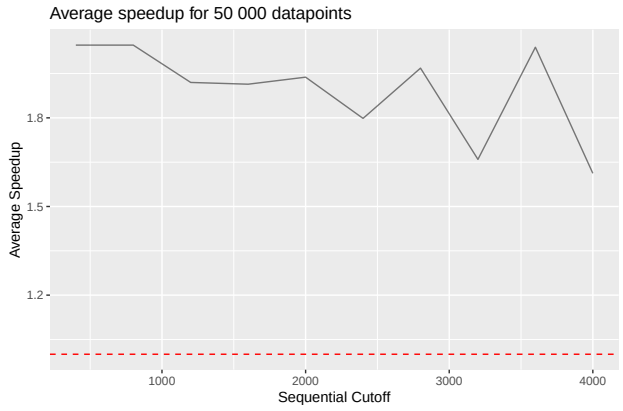

Figure 20

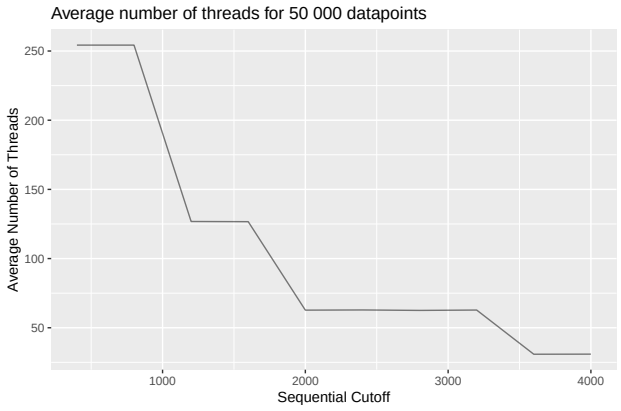

Figure 21

For an input size of 50 000, referring to *Figure 19* and *Figure 20*, it is clear that the parallel program outperformed the sequential program.

The optimal sequential cutoff appears to be at either 400 or 800. For these values, the parallel program achieved a speedup of approximately 2.05x, using 250 threads.

When contrasting these results to the Unix machine, not only is the sequential cutoff less for the Windows machine but the speedup is far superior (2.05x vs 1.1x). However, when comparing *Figure 16* and *Figure 4*, it is clear that the Unix machine in faster in general, due to the extra processing power that is possesses.

Similar to the Unix case, for this data size, we would suggest using the sequential program. Using parallelism doubled the speed of the program, however, this only equates to an approximate saving of 42ms - a very negligible saving. If it had achieved the same speedup, but with a larger data set, then the parallel program would have been chosen.

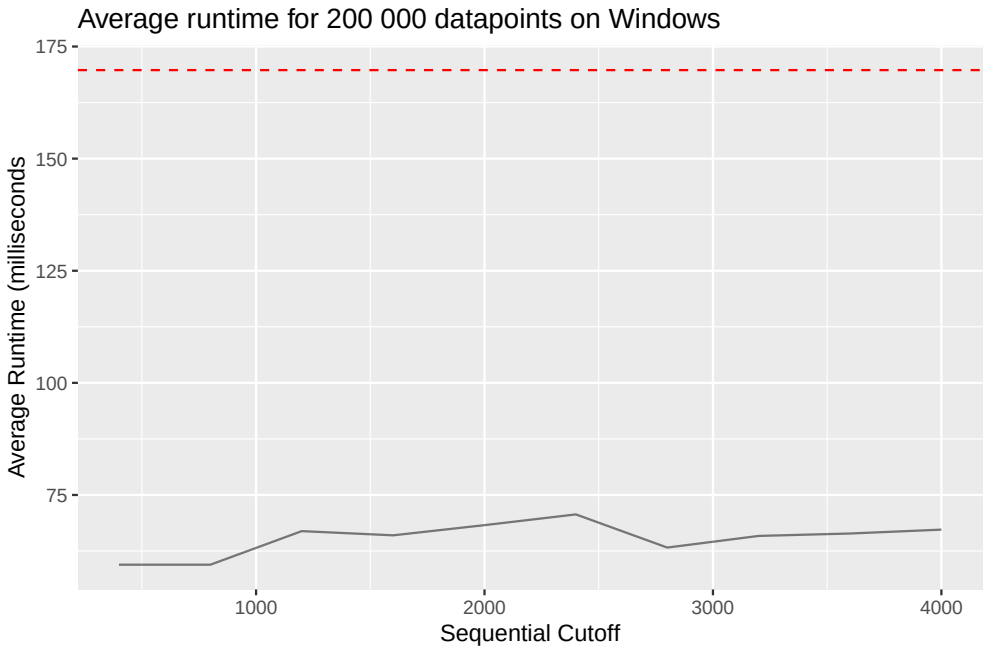

Figure 22

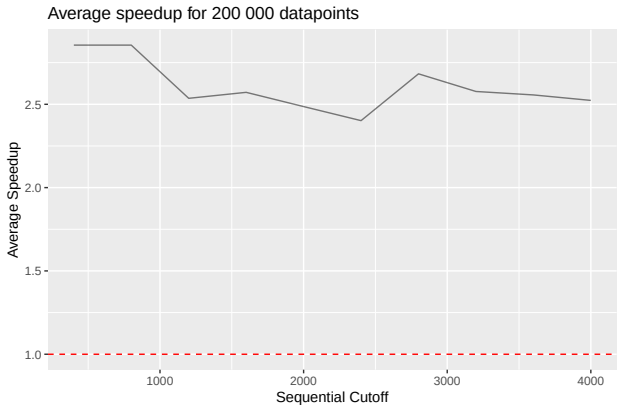

Figure 23

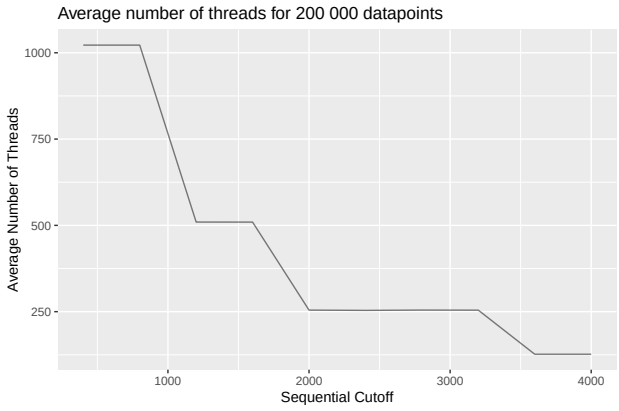

Figure 24

For 200 000 data points, we are now seeing a more significant improvement in the performance of the parallel program.

From *Figure 22* and *Figure 23*, the optimal sequential cutoff is either 400 or 800. This corresponds to a speedup of approximately 2.9x with 1050 threads utilized.

When comparing this to Unix (*Figure 7 & 8*), we see a similar trend to before. On the Windows machine, the sequential cutoff is lower and the speedup is considerably higher.

Although the speedup is approaching 3x, the parallel program is saving a maximum of approximately 107.5ms for this data size - just over 0.1s. This is not significant. Again, if the run times were considerably higher (on a larger data set or a more computationally expensive task) then this would be a significant speedup.

Thus, the sequential program is preferred again.

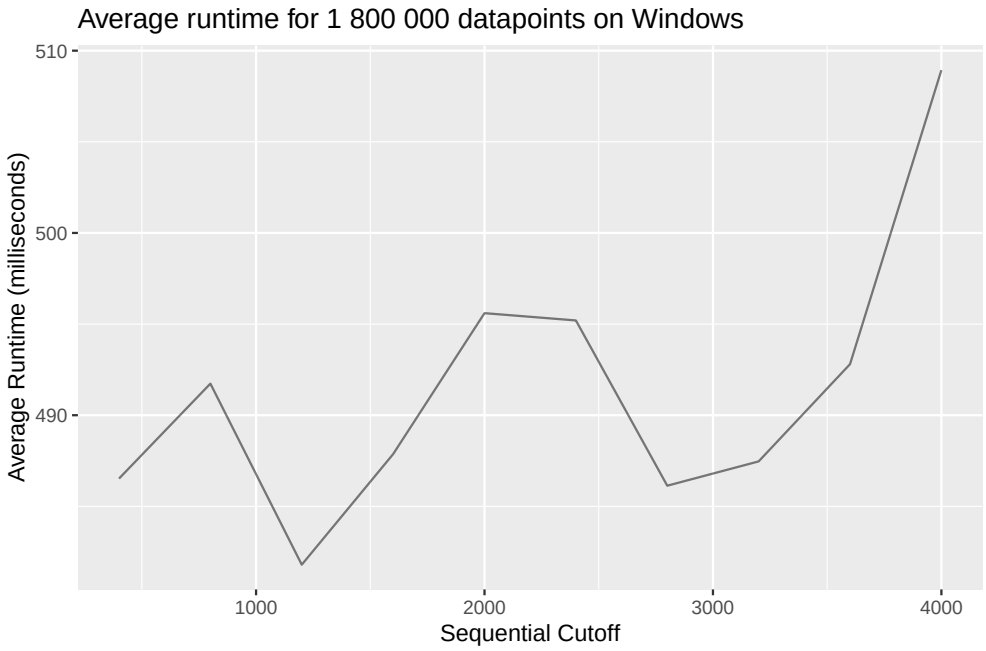

Figure 25

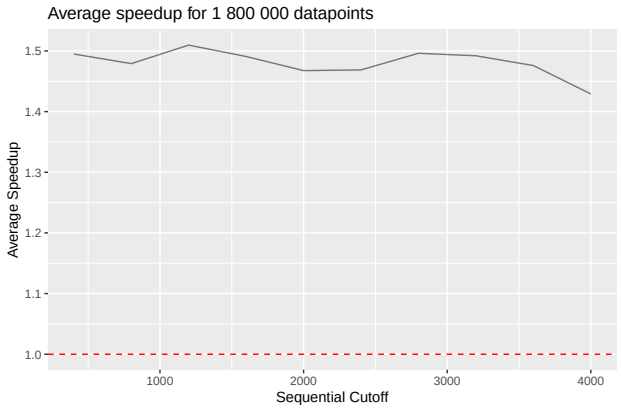

Figure 26

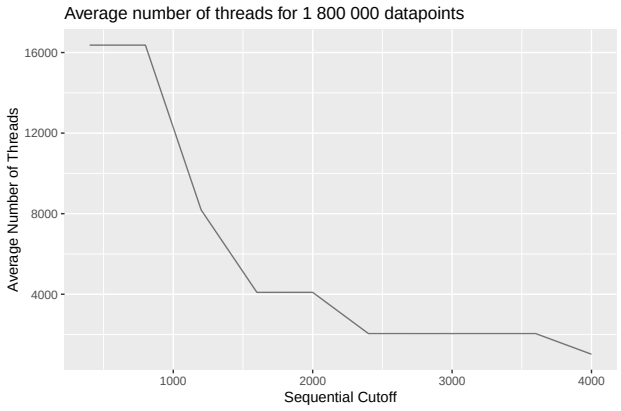

Figure 27

Finally, we will now examine the performance for 1 800 000 data points.

In *Figure 25*, the average sequential run time of 727ms was omitted because it distorted the scale of the graph severely. However, this is a sign of the improvement that the parallel program produced.

The optimal sequential cutoff is 1200. However, at this cutoff, we achieved a speedup of only approximately 1.51x, far less than we expected given the prior improvement in the performance of the parallel program for increasing data set sizes. So despite the parallel program saving a maximum of approximately 242ms, which seems like an improvement compared to the prior data set size, the speedup revealed that the parallel program actually performed worse than the prior input size.

The optimal number of threads used were approximately 8000.

Thus, because the speedup was only 1.51x and the parallel program only saved about 0.2s, the sequential program is yet again preferred for this data size.

GENERAL RESULTS

We will now briefly compare the Unix and Windows results.

Clearly the Unix machine was able to run the program faster for all input sizes, all sequential cutoffs and sequentially. This makes sense because the Unix machine has a faster processor and more cores than the Windows machine.

Interestingly, the optimal sequential cutoff values seemed to be lower for the Windows machine than the Unix machine. This means that the Windows machine generally performs better with more threads than the Unix machine, for the same input-data size. This seems rather counter-intuitive because you would expect a computer with more cores to be able to outperform a computer with fewer cores when using more threads as the system can delegate the threads more efficiently when there are more cores.

For the Windows machine, this could be due to the low clock speed, approximately only 2GHz. Presumably, the sequential part of the program is computed so slowly that, up to a certain point, the more threads utilized improves the run time (i.e. it is worth creating more overheads as minimizing the sequential component of the program outweighs this cost).

For the Unix machine, the higher sequential cutoff values could be due to its higher clock speed of 3.6GHz. This means that it could afford to have to compute more sequential operations, because of the superior speed that these operations took to compute.

Finally, it was apparent that the Windows machine had a larger speedup than the Unix machine. Again, we will conclude that this is due to a combination of the different clock speeds between the two machines and the low sequential work associated with this task.

For both machines, the speedup was relatively small due to the lack of computational work required by the sequential component for this task. Now we will compare the two machines:

The Windows machine has a lower clock speed, and thus you would expect the effects of parallelism to be greater when compared to the Unix machine. This is because minimizing the sequential work done by the Windows machine will be more beneficial than on the Unix machine as you will be penalized less for the more computationally slow (on the Windows machine) sequential component.

The Unix machine has a higher clock speed, and thus you would expect the effects of parallelism to be less when compared to the Windows machine. This is due to the Unix machine being able to run its sequential code faster than the Windows machine (due to the higher clock speed) leaving little room for improvement by the parallel program.

Finally, we will compare the maximal speedups to the theoretical speedups on the Windows machine and the Linux machine.

The maximal speedup for the Windows machine was 2.9x for a data size of 200 000. The theoretical speedup for this data size is $200000/log(200000) = 16385.29$x. Clearly the speedup obtained is no where near this value.

The maximal speedup for the Unix machine was 1.3x for a data size of 1 800 000. The theoretical speedup for this data size is $1800000/log(1800000) = 124971.4$x. Clearly the speedup obtained was also no where near this value.

However, the theoretical speedup calculation assumes that the computers have an infinite number of processors, and the Linux and Windows machines only have 2 and 4 processors respectively. Thus, you would expect the speedup of the program to be substantially less than the theoretical speedups quoted above. Furthermore, the theoretical speedup does not account for the overheads associated with creating the infinite number of threads to occupy the infinite processors. Thus, there is more support that the speedup should be substantially less than the theoretical.

Extension 1

In an attempt to see a significant speedup in the parallel program compared to the sequential program, we altered our code in order to create more sequential work - without changing the method for the task. Instead of simply summing the 8 surrounding values in order to calculate the local wind average, we used the following result $x = e^{ln(x)}$, and applied this four times to each value before summing. This increased the sequential work significantly.

In order to maximize the chances of a significant speedup, we applied this more computationally expensive approach to an input data size of 20 000 000, on the Unix machine (which has more cores). The results are seen below:

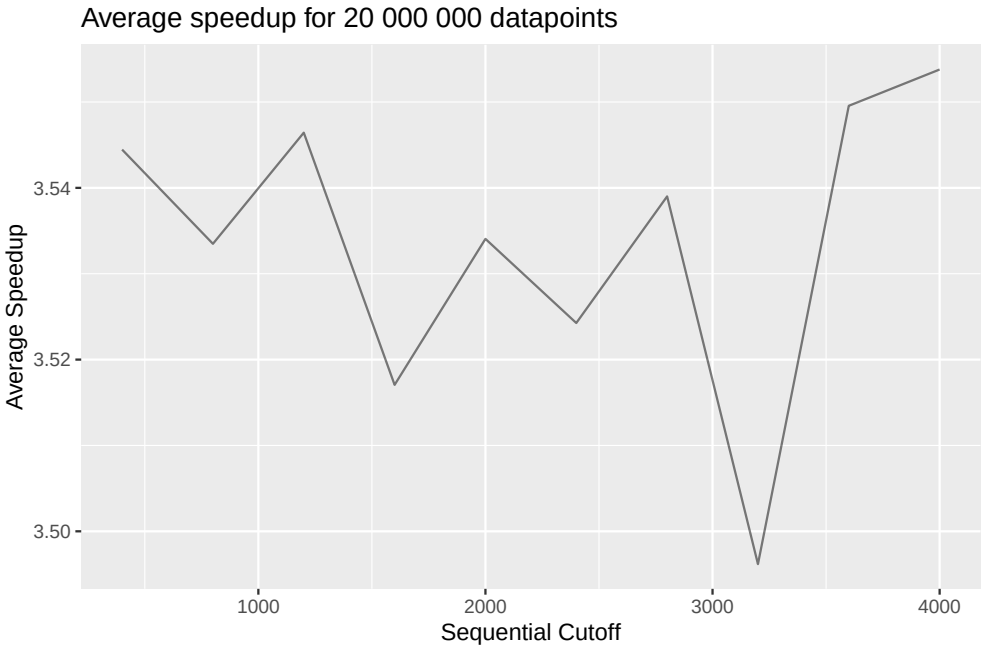

Figure 28

Now we are seeing a maximal speedup of approximately 3.57x (over our range of sequential cutoffs). When you compare this to *Figure 14*, when we tested the same sized data set on the Unix machine without adding extra computational work, we only saw a maximum speedup of approximately 1.07x. Adding the extra work more than tripled the speedup achieved.

This supports the argument that parallelism would be more appropriate when applied to a task with more sequential work - our task did not have sufficient computational work for the effects of parallelism to really take effect.

Conclusions

Unfortunately, the results from the experiments were fairly disappointing.

There definitely seemed to be a general relationship between input size and parallel performance. For all of the data sets besides the largest for both machines, the larger the data set size, the faster the parallel program ran when compared to its sequential counterpart. However, despite these increases in performance, it was concluded for all tests that the gain of using parallelism was not significant enough to warrant using parallelism over sequential programming because the run time saving was not significant enough and due to the extra simplicity to read and write sequential code.

Furthermore, in the tests where the speedup was substantial, the run time saving was not. Thus, if the same speedup was observed for larger run time values, then the parallel program would have been preferred. However, we saw that both computers seemed to reach a point whereby the data sets seemed to become "too large", resulting in the performance of the parallel program over the range of sequential cutoffs to worsen.

The maximum speedup obtained by the parallel program was 2.9x for a data size of 200 000, on the Windows machine. In general, it seemed that the Windows machine had a larger speedup than the Unix machine. We decided that this was due to the difference in clock speeds between the two machines, as discussed in *General Results*. We saw that the optimal speedups for both machines were substantially slower than the theoretical speedup, but that was expected because the theoretical speedup uses assumptions that were clearly violated - namely, that we have an infinite number of processors, and that there are no overheads when creating threads.

As we had expected, the optimal sequential cutoff varied by machine and by input size, where the Windows machine had a lower cutoff value in general compared to the Unix machine. The specific values of the sequential cutoffs can be seen in the *Results and Discussions* section. It seems fairly clear that we cannot specify an optimal sequential cutoff for this problem, due to the variation that was seen between machines and input sizes.

The optimal number of threads used by each architecture is directly related to the sequential cutoff as the lower the sequential cutoff, the more threads used (and vice versa). Thus, the conclusion for the optimal number of threads mimics that of the optimal sequential cutoff. There was no optimal number of threads for each architecture because of the variability in this number for each input size. However, we did see that the Windows machine optimally used more threads than the Unix machine.

Extension 2

To further extend this assignment, we decided to produce the whole pdf report through Rmarkdown. The program reads in all of the test output text files, making use of data frames, and then uses these data frames with the `ggplot` library to produce the plots seen in this report.