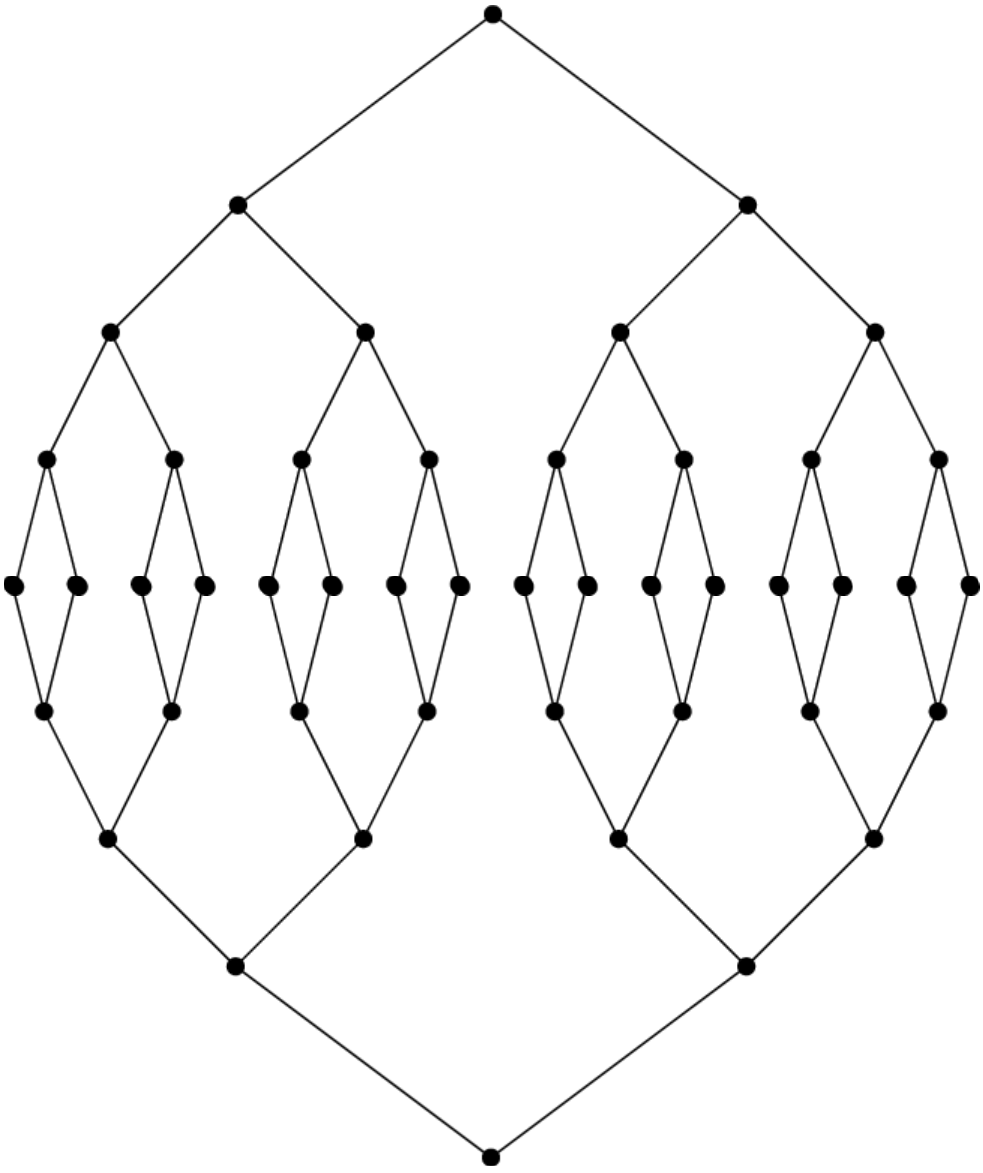
Fork/Join Framework Analysis



Rahul Mistri

CSC2002S
Assignment 3

INTRODUCTION

Context

For a long time, programming was entirely serial and everything was done using a single processor. Moore's law roughly states that the number of transistors on a chip doubles roughly every 18 to 24 months. This was great because if a program was slow, one could just wait until a faster processor is developed in order to run the program faster. But due to heat, power and memory access constraints, it is no longer possible. inefficient algorithms remained inefficient until parallel programming became popular. In order to circumvent the limitations holding back Moore's Law, processing power was increased by adding more cores to the processor. Now parallel programming is used almost everywhere as it is one of the only ways to increase the speed of a program. However it is not an easy task, parallel algorithms are harder to write than their serial counterparts, and it is even harder to ensure they are correct and efficient.

Aim

Using the Java Fork/Join framework and a divide-and-conquer algorithm, the classification of weather data is parallelized and analysed for a range of data sizes and sequential cutoffs to determine if parallelization is worth using. If it is, questions regarding the ideal range of data sizes, sequential cutoffs and number of threads will be investigated.

Algorithm Overview

The main parallelization will follow a divide-and-conquer strategy and make use of the fork/join framework offered by Java, with the crux of the algorithm described later.

Expected Performance

Given that testing will occur on processors with 8 cores and 4 cores, Amdahl's law suggests we should expect a speed up of at most 8 and 4 respectively. Given that overheads are present, it is extremely unlikely that it will be 8 and 4, but it will not be more than that.

METHOD

Algorithm

There are 2 main algorithms to this project, this section will give a quick overview of each one.

1. The Local Average calculation

   Given the (i, j, t)th position in the array of size (x,y,z), note that 0<i<x, and 0<j<y.

   sum, neighbours = 0

   for (integer a = max(0, i-1), i<min(x,i+2),i++)

   | for(integer b = max(0, j-1), i<min(y,j+2),j++)

   | | sum = sum + vector[a,b,t]

   | | Neighbours++

   | |_____

   |_____

   Average = sum/neighbours

   Although there is a nested for loop, it can only run for a maximum of 3 times on each loop, or 9 total iterations. The t parameter remains constant as the local average is only calculated across one time stamp, hence the need for only two for loops. Min and Max functions enable the algorithm to account for the boundary cases like if the central element is in the corners or on the edges.

2. The Parallel portion

   Given a min position and a max position of a 1 dimension array (that is equivalent to the 3 dimensional array)

   If (max-min < SEQUENTIAL_CUTOFF)

   The local average is calculated and classified

   For each element in the 1d array from min to max, the vectors are summed

   Else

   two new parallel threads are created from min to min+max/2 called left and (min+max)/2 to max called right.

Left is forked, right is computed and when it is completed, left joins.

(This is the recursive divide and conquer algorithm that will speed up the program.)

Approach

The general idea was to collate runtime data for each data size by recording the runtimes for the serial program, and then recording the runtimes for numerous sequential cutoffs for the parallel program. This was repeated for numerous data sizes. In general, the sequential cutoffs followed the geometric series of 10^x ranging from below the smallest data size to above the greatest data size. This ensured each data size had a chance to run sequentially using the parallel algorithm, to illustrate the effects of the overheads that parallelization has.

Validation and Testing

The algorithm was validated by running it on numerous small inputs with simple to calculate averages by hand. By using the fork/join framework, no race conditions occurred and the general thread was designed to refer only to its own variables, and not shared variables ie. there were no static totals to update. Furthermore, the output data was compared to Amdahl's Law, and all data obeyed and followed it. The parallelization, as shown later, also provided speed up hence being efficient.

In order to get accurate timing, the first 10 tries were discarded so that the data was in cache, after which, the average time over 500 tries was taken to ensure an accurate time.

The project was tested on two architectures, which shall be called Architecture 1 and Architecture 2 in the rest of the report.

Architecture 1:

Intel® Core™ i7-4700MQ CPU @ 2.40GHz × 8 (8 cores)

Architecture 2:

Intel® Core™ i5-7400 CPU @ 3.00GHz × 4 (4 cores)

RESULTS

Figure 1 (Architecture 1)

	10	100	1000	10000	100000	1000000	10000000	Serial
200000	0.00804	0.0055	0.00334	0.00278	0.00396	0.01072	0.0111	0.00992
800000	0.01018	0.00944	0.00908	0.00884	0.00908	0.03946	0.04018	0.03852
1800000	0.0212	0.01944	0.0182	0.01754	0.01868	0.04462	0.08588	0.0843
3200000	0.03832	0.0338	0.0307	0.03386	0.03136	0.0438	0.15306	0.1504
5000000	0.0579	0.0527	0.04736	0.04588	0.0476	0.05158	0.2318	0.22624

Figure 1 is a look-up table to find the time taken on Architecture 1 to process a normal data size (row) for specific sequential cut offs or in serial (column).

Figure 2 (Architecture 2)

	10	100	1000	10000	100000	1000000	10000000	Serial
200000	0.00586	0.00628	0.00434	0.00402	0.00462	0.01104	0.01106	0.01474
800000	0.01134	0.01034	0.0112	0.01072	0.0108	0.04034	0.0402	0.03648
1800000	0.0257	0.02264	0.02228	0.02196	0.02276	0.04564	0.08714	0.07966
3200000	0.04516	0.0418	0.03922	0.03856	0.03872	0.04134	0.15366	0.14412
5000000	0.07336	0.06388	0.06118	0.0595	0.0604	0.073	0.23868	0.21994

Figure 2 is a look-up table to find the time taken on Architecture 2 to process a normal data sizes (row) for specific sequential cut offs or in serial (column).

Figure 3 (Architecture 1)

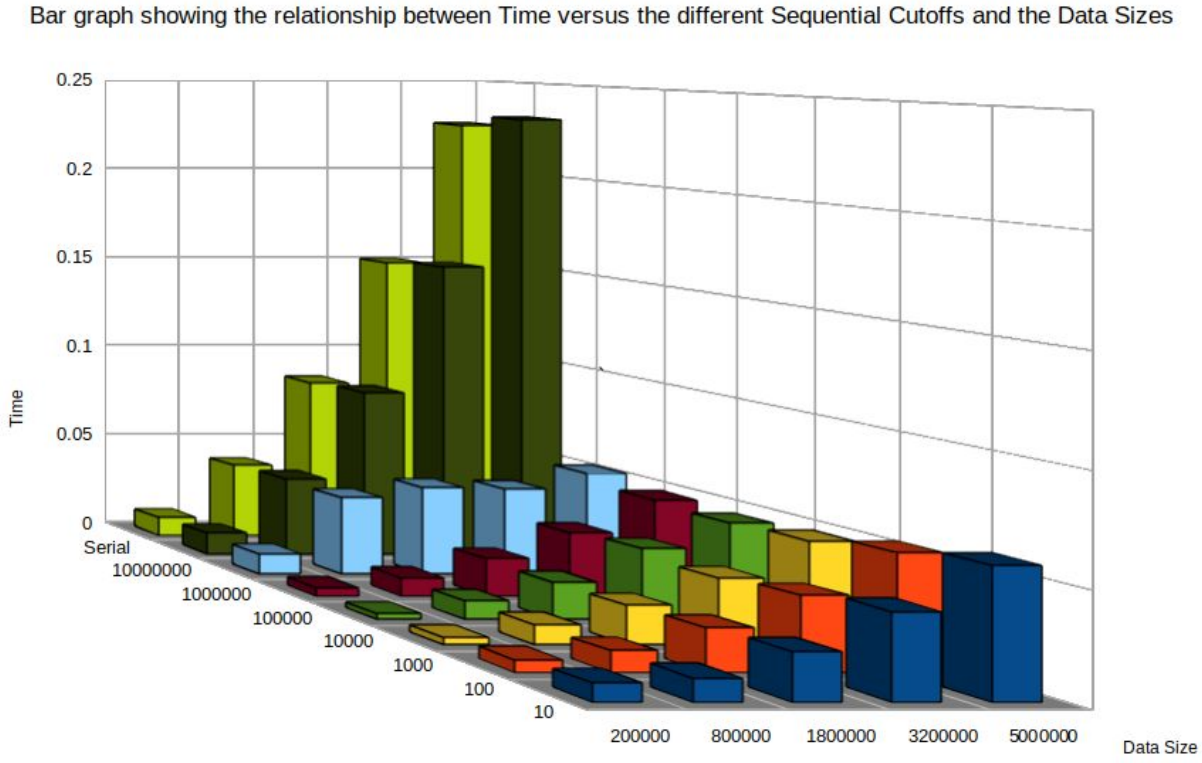


Figure 4 (Architecture 1)

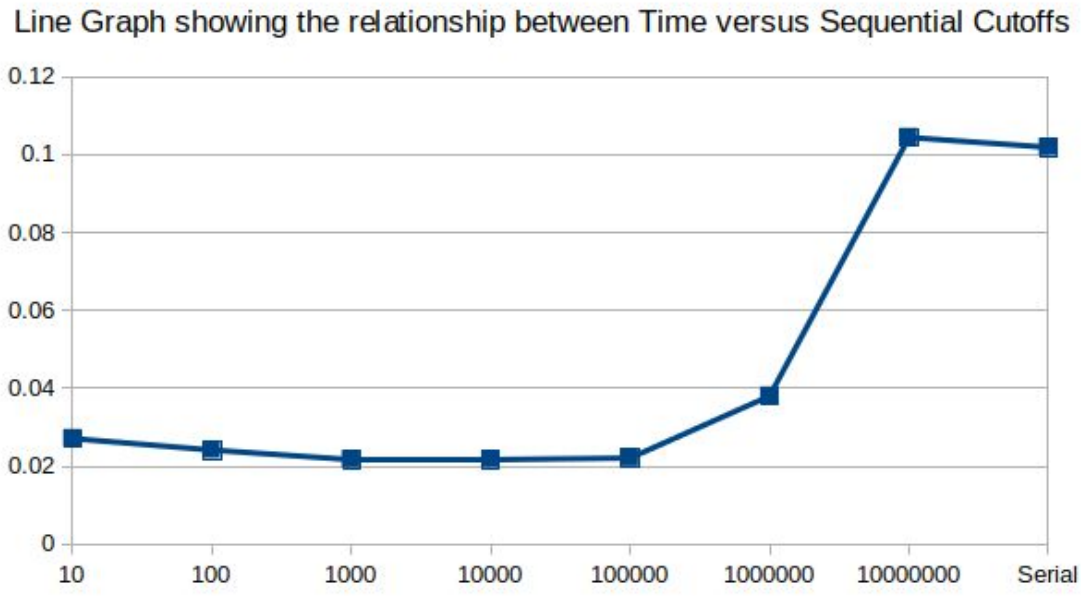


Figure 4 shows the relationship of the average time (of the data sizes) against the different sequential cutoffs.

Figure 6 (Architecture 1)

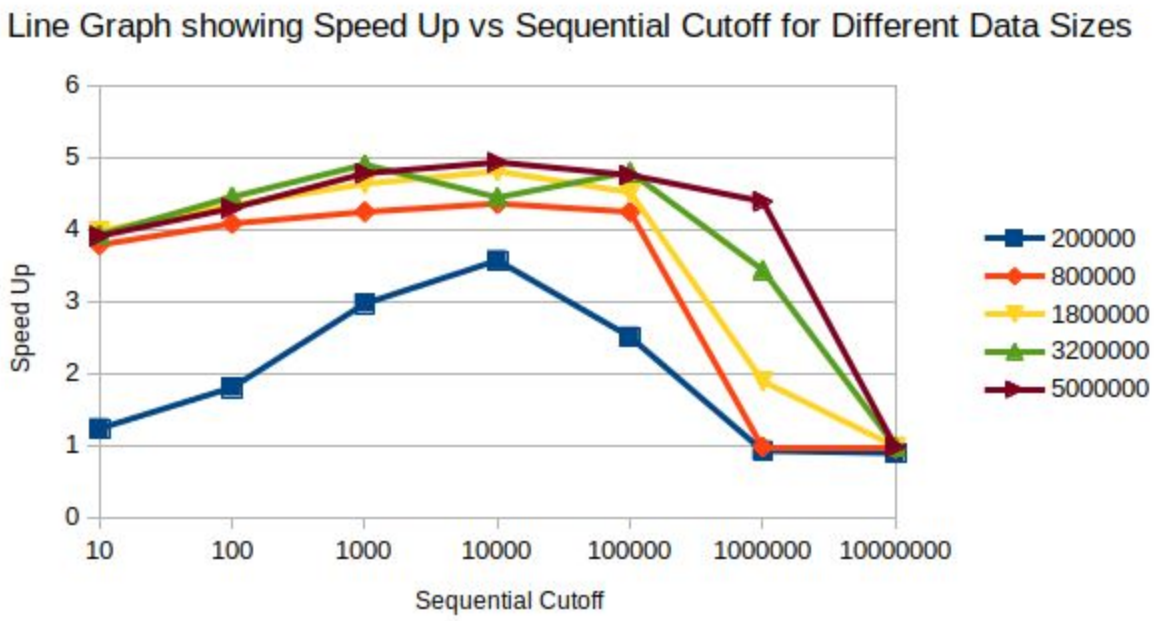


Figure 6 shows the speed up for each data size against the sequential cut offs

Figure 7 (Architecture 2)

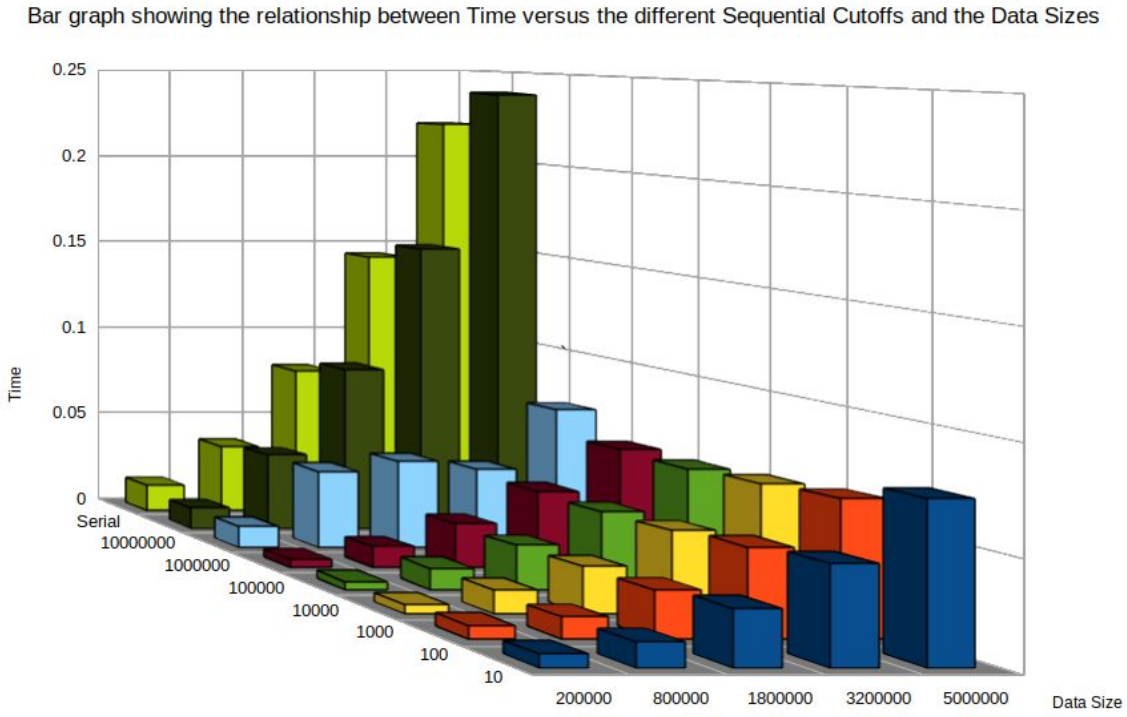


Figure 8 (Architecture 2)

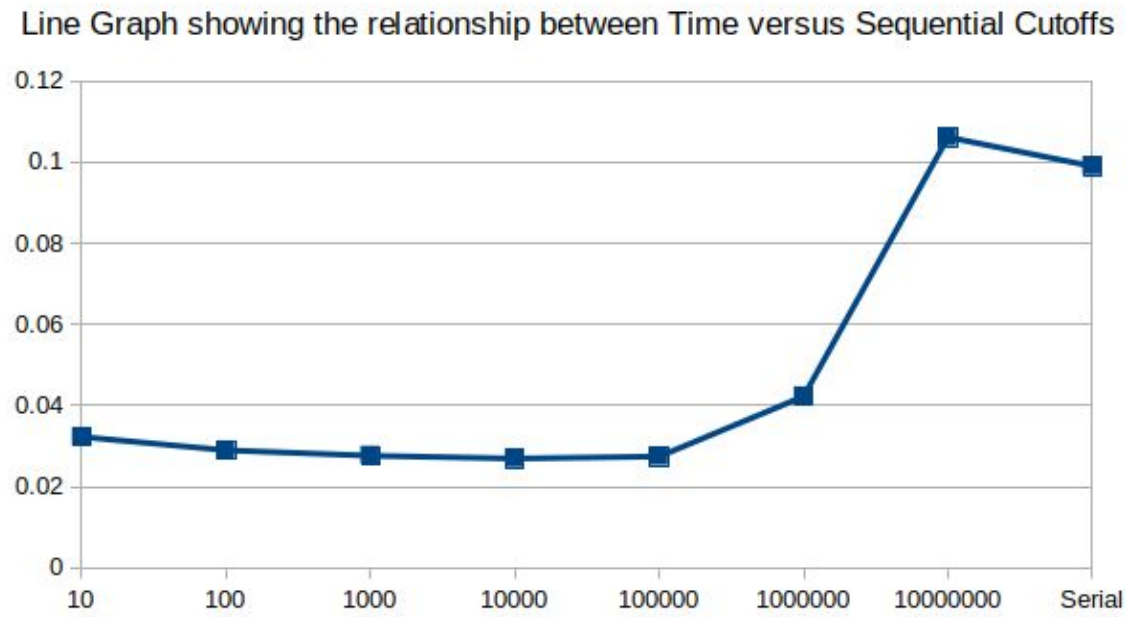


Figure 8 shows the relationship of the average time (of the data sizes) against the different sequential cutoffs.

Figure 9 (Architecture 2)

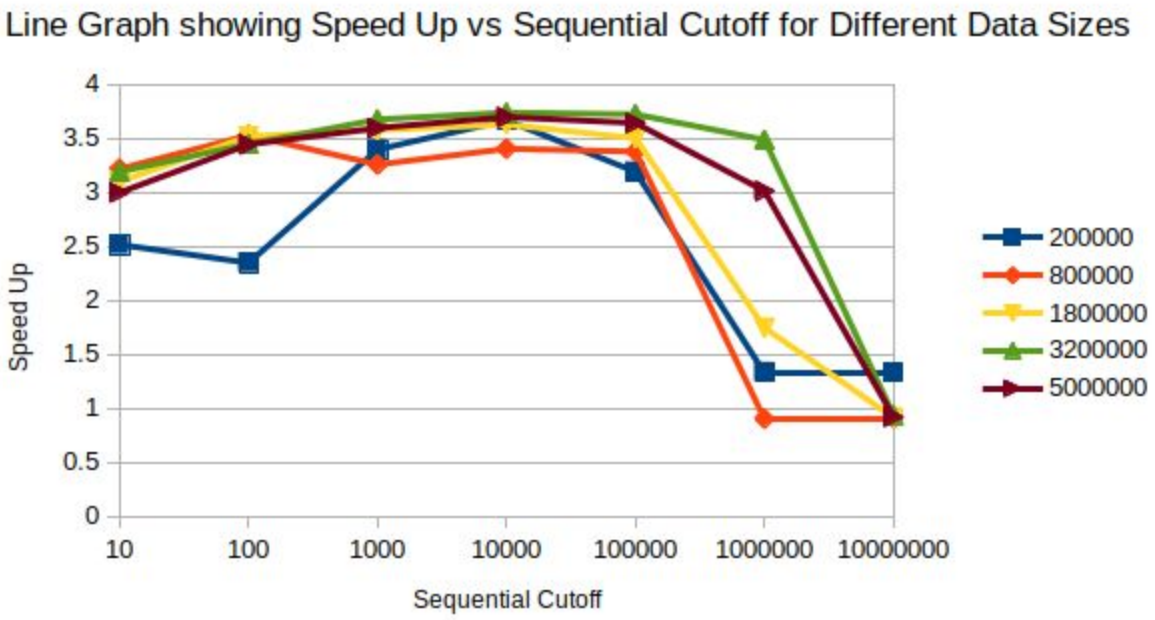


Figure 9 shows the speed up for each data size against the sequential cut offs

DISCUSSION

When processing large amounts of data, figures 3 and 7 clearly show the benefit of making use of parallelization and multithreading to speed up problems like this one. More specifically, problems that have solutions that are commutative, like the summation of an array of values is definitely suited to parallelization as they are not affected by the non-deterministic nature of parallelization.

Referring to figures 6 and 9, the dark blue line is for the smallest data size, and the green and maroon lines are for the larger data sizes. The speed up experienced by the larger data sizes is larger than that of the blue line. Although parallelization seems to suit almost all data sizes tested, the data indicates that larger data sizes tend to have much better speed up than their smaller counterparts.

Amdahl's law states: $$Speedup = \frac{1}{(1-p) + p/N}$$

Where p is the proportion of the program that is parallelizable with 0<p<1, and N being the number of processors. In architecture 1, where N=8, as p tends to 1, the maximum speedup possible is 8. In architecture 2, where N=4, as p tends to 1, the maximum speed up is 4. In other words, given N processors, the maximum speedup that can be reached is N. Due to the overheads of the threads, as well as heat or power issues, it is unlikely and basically impossible to get a speedup of N. In architecture 1, the maximum speed up obtained was 5, and in architecture 2, the maximum speed up obtained was approximately 3,5. These were both at least 50% of the maximum attainable speedup.

According to figures 4 and 7, which took the average times across the data sizes per sequential cutoff and plotted it against the sequential cutoffs, the optimal sequential cutoff is 10 000, which correlates to figures 3 and 7, where all times are lowest at sequential cutoff equal to 10 000. It is clear in figure 8 that when the sequential cut off is greater than all data sizes, the overheads of the threads cause it to be less efficient than

its serial counterpart.

Looking at figures 6 and 9 again, we can clearly see that smaller data sizes do not fare well against large sequential cut offs, but large data sets do. This is because as the sequential cutoff increases, smaller data sizes are split into fewer threads until they are run by a single thread. Referring to figures 1 and 2, the smaller data size takes as much, and most often even more time than its serial counterpart when the sequential cutoff is greater than its data size. This is due to the overheads of the thread on which it runs on. The ideal sequential cutoff for any data size must be less than the size of the data.

Looking at figures 6 and 9, maximum speed up is attained when the sequential cutoff is between 10 000 and 100 000, suggesting that the average number of threads recommended for both architectures are approximately 100-500 threads.

EXTRA CREDIT (INVESTIGATING VERY SMALL AND VERY LARGE DATA SIZES)

Figure 10

	10	100	1000	10000	100000	1000000	Serial
8000	8.20E-04	6.60E-04	8.20E-04	8.80E-04	8.20E-04	8.80E-04	7.00E-04
32000	0.0014	9.80E-04	5.40E-04	0.00106	0.00198	0.002	0.00184
72000	0.0014	0.00122	0.00104	0.00132	0.0053	0.00464	0.00396
128000	0.0022	0.00192	0.00178	0.00208	0.0038	0.00772	0.00706

Figure 10 is a look-up table to find the time taken on Architecture 1 to process a small data sizes (row) for specific sequential cut offs or in serial (column).

Figure 11

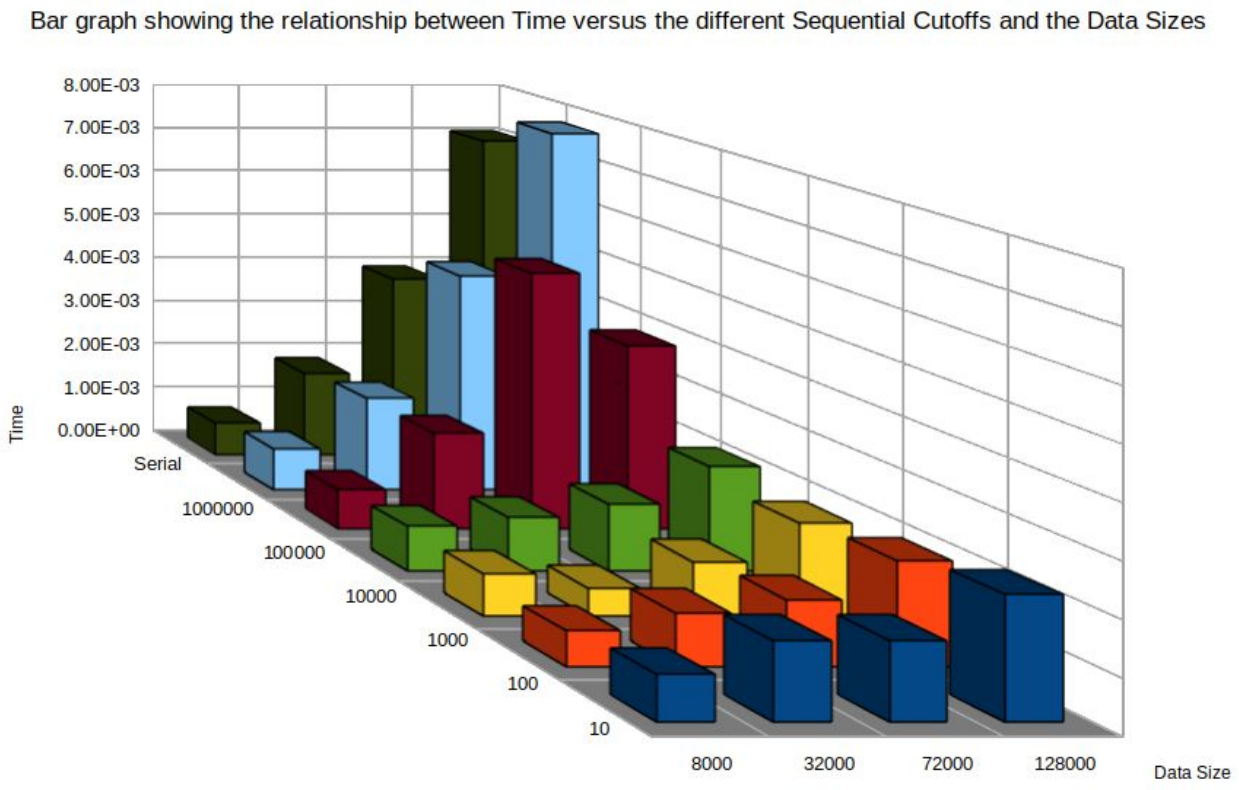


Figure 13

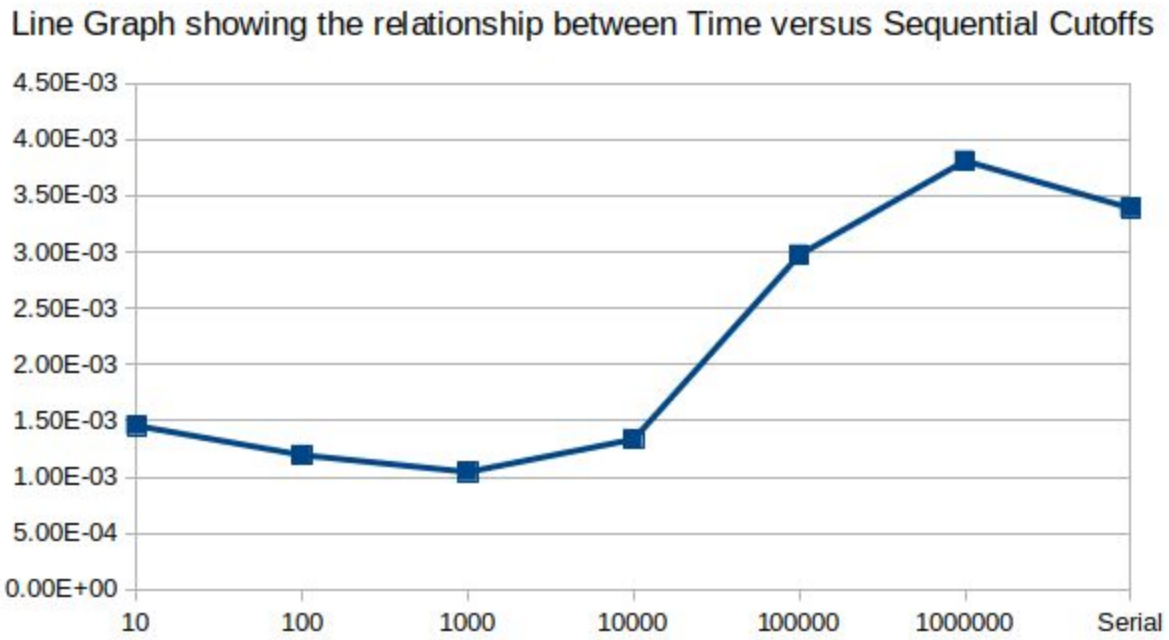


Figure 13 shows the relationship of the average time (of the small data sizes) against the different sequential cutoffs.

Figure 14

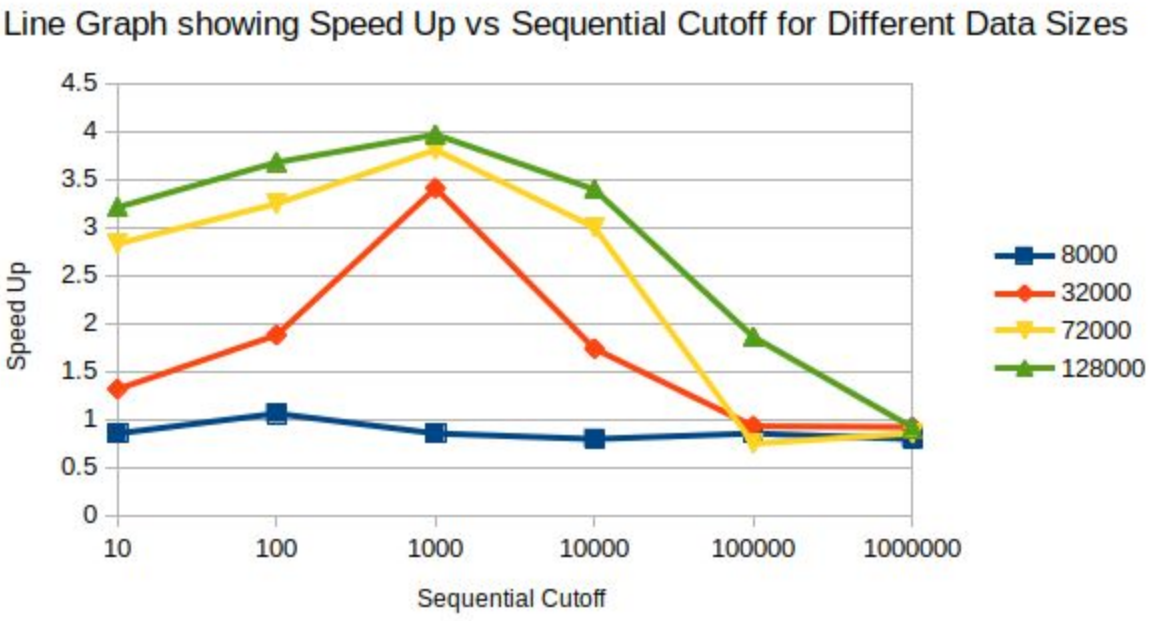


Figure 14 shows the speed ups vs sequential cutoffs for small data sizes.

Figure 15

	100000	1000000	10000000	100000000	Serial
11250000	0.1184	0.11576	0.24498	0.50772	0.49662
20000000	0.20968	0.1916	0.22142	0.88962	0.88164
31250000	0.32266	0.30304	0.3512	1.40264	1.36476
45000000	0.49912	0.4439	0.42278	2.00386	1.99082

Figure 15 is a look-up table to find the time taken on Architecture 1 to process large data sizes (row) for specific sequential cut offs or in serial (column).

Figure 16

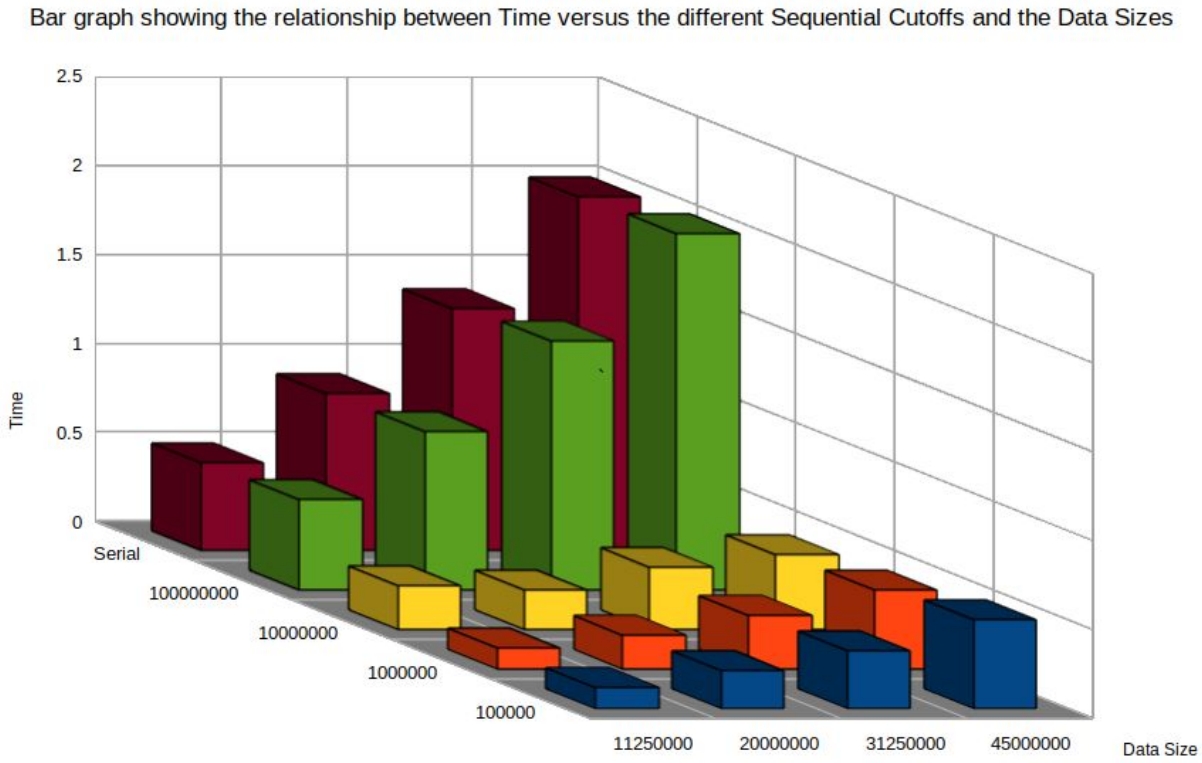


Figure 17

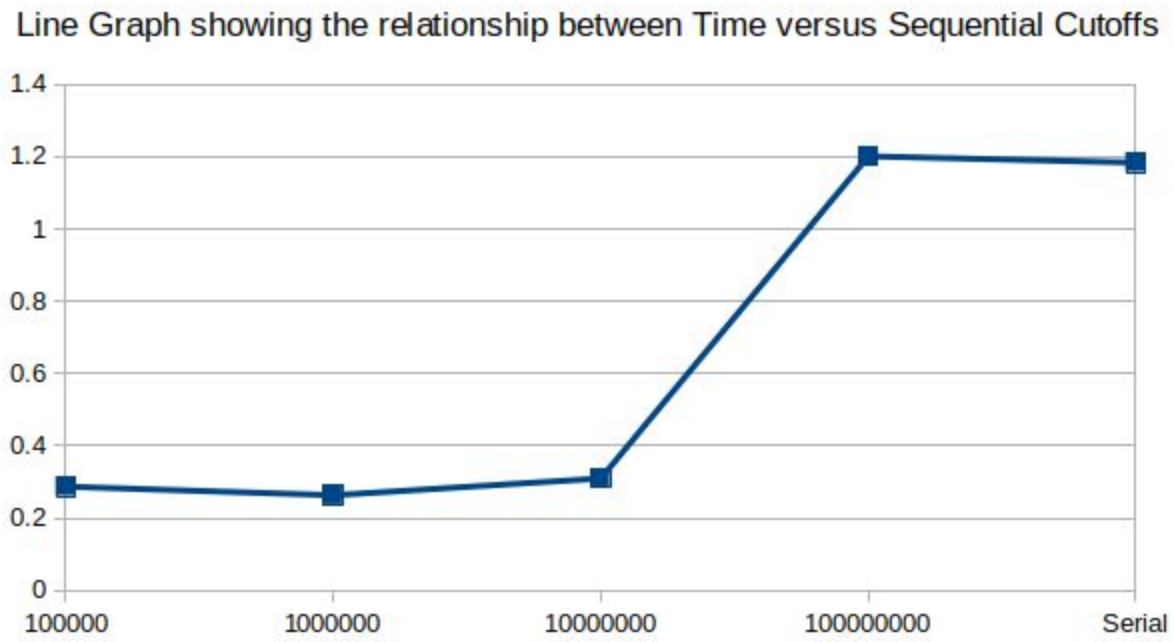


Figure 17 shows the average time taken across each sequential cutoff (over the data sizes) against the different sequential cutoffs.

Figure 18

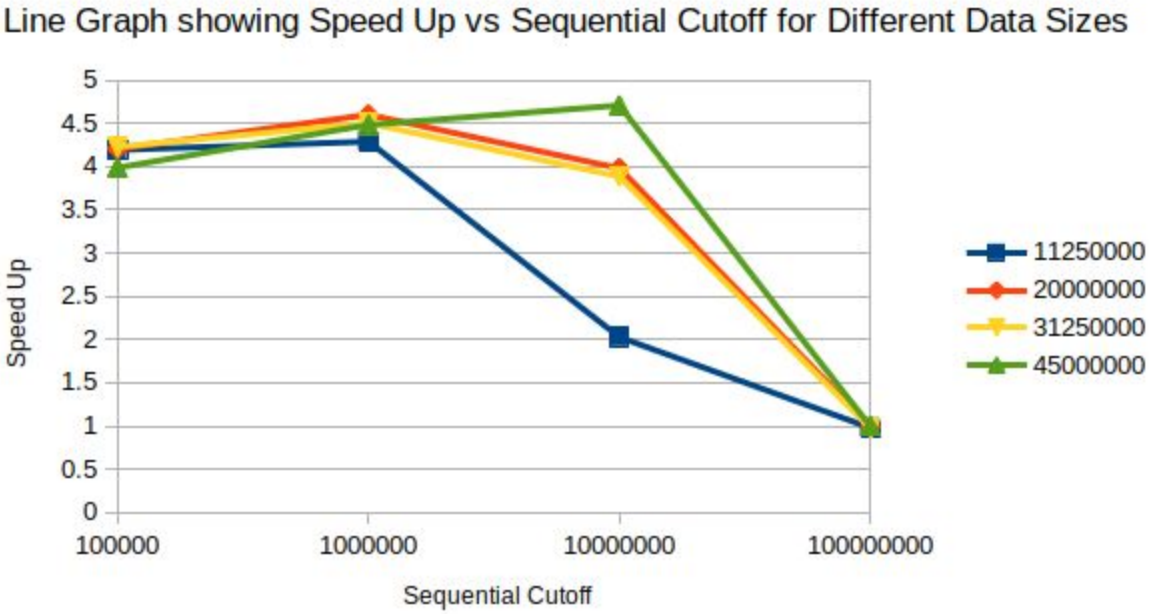


Figure 18 shows the the speed ups for each data size against the different sequential cutoffs.

FURTHER DISCUSSION

The claims made in the previous discussion are backed by the data presented above, as even for very small and very large data sizes, parallelization sees significant speed up.

Figures 14 and 18 support the claim that larger data sizes have even better speedup compared to smaller data sizes as well. They also show that the optimal sequential cutoffs for small data sizes is between 100-1000 for small data, and between 1000000 -10000000 for large data sizes. From before, on these architectures, the optimal sequential cutoff for a data size X is between X/1000 to x/10.

The maximum speedup attained is 4.7 for large data sizes, and between 3.5 and 4 for small data sizes. This supports the fact that for small data sizes, the cost of overheads take a toll on the achievable speed up, whereas with large data sizes, the cost of those overheads is negligible compared to the benefits of multithreading, hence they have larger speedup.

CONCLUSION

In conclusion, the following assumptions can be made:

- For problems that have answers that are not order-dependent during calculation, ie they are commutative, the parallelization is a well-suited approach to take
- Parallelization gives speed up on all data sizes, provided the sequential cutoff is chosen carefully
- Although speedup is experienced by all data sizes, larger data sizes gain more from parallelization as they do not suffer as much from the overhead costs of the many threads that smaller data sizes do
- The sequential cutoff must be less than the data size, and based on the above evidence, the optimal sequential cutoff for a data size X is between X/1000 to x/10.

GIT log

899aa0dac9eb84505874090bef19b742d54b2433 added java docs, made runnable from CLI

756baaecbfdc6d64c33905ddc62fa4dd56e4703b testing done

3486a8107e88db2e963aba918e98f0a5635a8472 parallel and sequential ready for testing

ce10fd2d18b2cbf5804c7c6a19569c3f0e902a57 all code finished, testing to begin next

e3916ac0ef9c09b93b01a2e7a13c5a1821d4c6ae sequential calc working

4f25f5e6849c5a091eb24619dd1edbd675986539 finished sequential calculator

388654712165c5e9bc36d0bd99d4731c3b108577 added Vector class, begun sequential calc

1afdf37daad43b6768a0968df0275bc297c47a10 Added CloudData.java

248f17c42ce9d9be272fd6d72d870ff3712e0c86 Added git ignore file