Introduction

Cross-correlation (also known as the sliding dot product) is a measure of similarity between two signals. It works by sliding one signal along in time, summing the products of the two signals at each point. The point of maximum correlation will be the point where the two signal are most similar in that the product of the aligned peaks and aligned troughs contribute largely to the sum.
The correlation algorithm described in the assignment can be quite expensive as it includes two nested for loops meaning a quadratic algorithm if the signals are of equal length. The algorithm is easily implemented in parallel as the correlation can be calculated by different threads summing for different shifts in time.

Methods

Three different versions of the algorithm were implemented, a serial version and two parallel versions. These were handled in the classes Serial.java, Parallel.java and ParallelSplit.java and Parallel2.java and ParallelSplit2.java.

Serial

This version runs the serial algorithm for a comparison of the speedup obtained with the use concurrent programming. It does not build the cross-correlation array but just keeps track of the position of the maximum correlation.

Parallel

This version uses the classes Parallel.java and ParallelSplit.java. It divides the received signal (outer for-loop) into new threads repeatedly until the size of the working array is less than the sequential limit. It does not build the cross-correlation array but just keeps track of the maximum correlation and its position in a Pair object. When the threads are joined, the Pairs can be compared and the bigger maximum is returned.

Parallel2

This version uses the classes Parallel.java and ParallelSplit.java. It divides the received signal (outer for-loop) into new threads repeatedly until the size of the working array is less than the sequential limit. It builds the cross-correlation array. When the threads are joined, the returned arrays are joined together using System.arraycopy() and when the whole array has been returned, the array is searched for the true maximum and its position.

The purpose of making the two parallel programs was that only the maximum needed to be found yet in the original algorithm, an entire array of the correlations is kept, using more resources. However, the second parallel program was programmed and run to see if this was indeed the case as well as to implement the original algorithm.

Max

The method to find the maximum of the correlated array was also implemented in parallel. It splits the array into different working arrays and searches sequentially in each thread, using Pair objects to compare as in the Parallel version of the program.

Timing

Timing of the programs was done using System.currentTimeMillis(), timing only the correlation.

Sequential Limit

The sequential limit can be entered when the program is run. Limits of 1000, 500 and 200 were used in testing.

Results Used

Five runs for each test were recorded. The average of the three fastest times were used for the results. This was done as some extreme times were recorded.

System Used for Testing
Macbook Pro, 13-inch, Mid 2012:
Processor 2,5 GHz Intel Core i5 with 4 cores.
Memory 4 GB 1600 MHz DDR3
Graphics Intel HD Graphics 4000 1024 MB

Validation
The signals were plotted using JFreeChart, shifting the transmitted signal forward in time by the offset amount. This showed the signals lining up and a correct correlation.

Results and Discussion

Sequential limits of 1000, 500 and 200 were used (in brackets).
The following tables split the results up by the program used. The average of the results only used the three fastest times. This was done to remove any extreme times that may have been recorded, allowing Java's Fork/Join time to "warm up" as well as removing strange results such as Parallel2 (1000) run 4 which could have occurred with the CPU being interrupted for other tasks.
The graphs are split according to the data size to compare the different programs.

Tables

Serial						
	Run Time (seconds)					Average
Size of Data	1	2	3	4	5	(3 Fastest)
10000	0.107	0.125	0.087	0.087	0.089	0.088
100000	8.753	8.193	8.345	8.172	8.154	8.173
1000000	827.607	822.195	822.652	825.440	823.501	822.783

Parallel (1000)						
	Run Time (seconds)					Average
Size of Data	1	2	3	4	5	(3 Fastest)
10000	0.093	0.040	0.042	0.041	0.042	0.041
100000	3.557	3.786	3.517	3.805	3.511	3.528
1000000	371.841	369.833	355.851	367.176	358.362	360.463

Parallel (500)						
	Run Time (seconds)					Average
Size of Data	1	2	3	4	5	(3 Fastest)
10000	0.155	0.047	0.042	0.044	0.044	0.043
100000	3.683	3.511	3.716	3.709	3.520	3.571
1000000	361.164	371.802	360.562	367.560	355.580	359.102

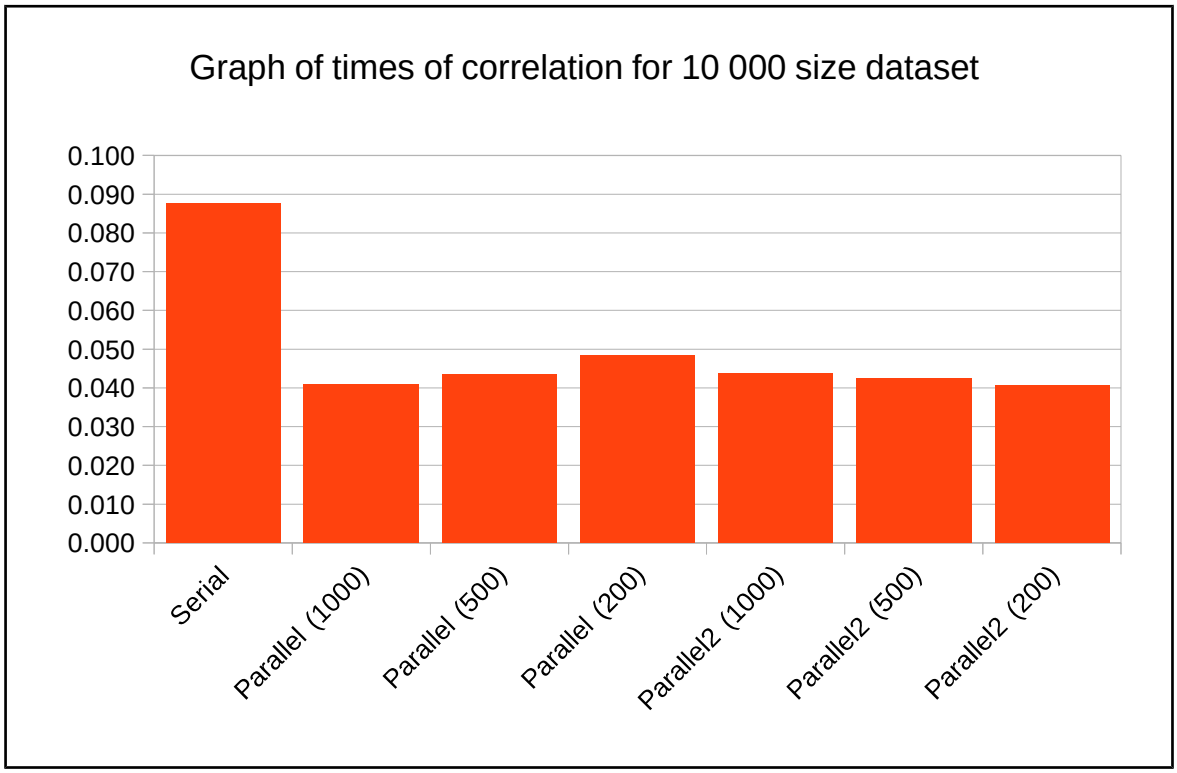
Parallel (200)						
	Run Time (seconds)					Average
Size of Data	1	2	3	4	5	(3 Fastest)
10000	0.096	0.045	0.050	0.050	0.052	0.048
100000	3.671	3.685	3.682	3.718	3.641	3.665
1000000	355.789	357.398	356.914	356.063	356.197	356.016

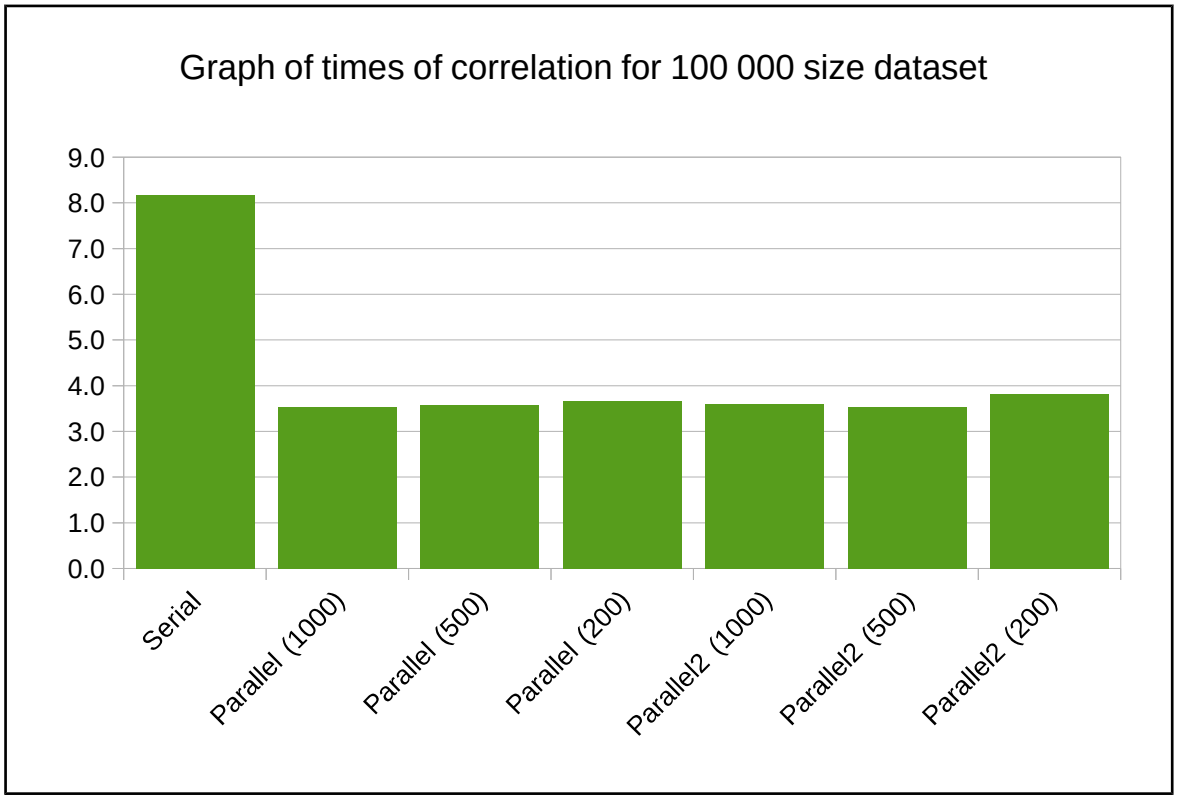
Parallel2 (1000)						
	Run Time (seconds)					Average
Size of Data	1	2	3	4	5	(3 Fastest)
10000	0.126	0.044	0.044	0.043	0.049	0.044
100000	3.937	3.783	3.564	3.514	3.706	3.595
1000000	386.209	381.161	380.265	401.532	380.911	380.779

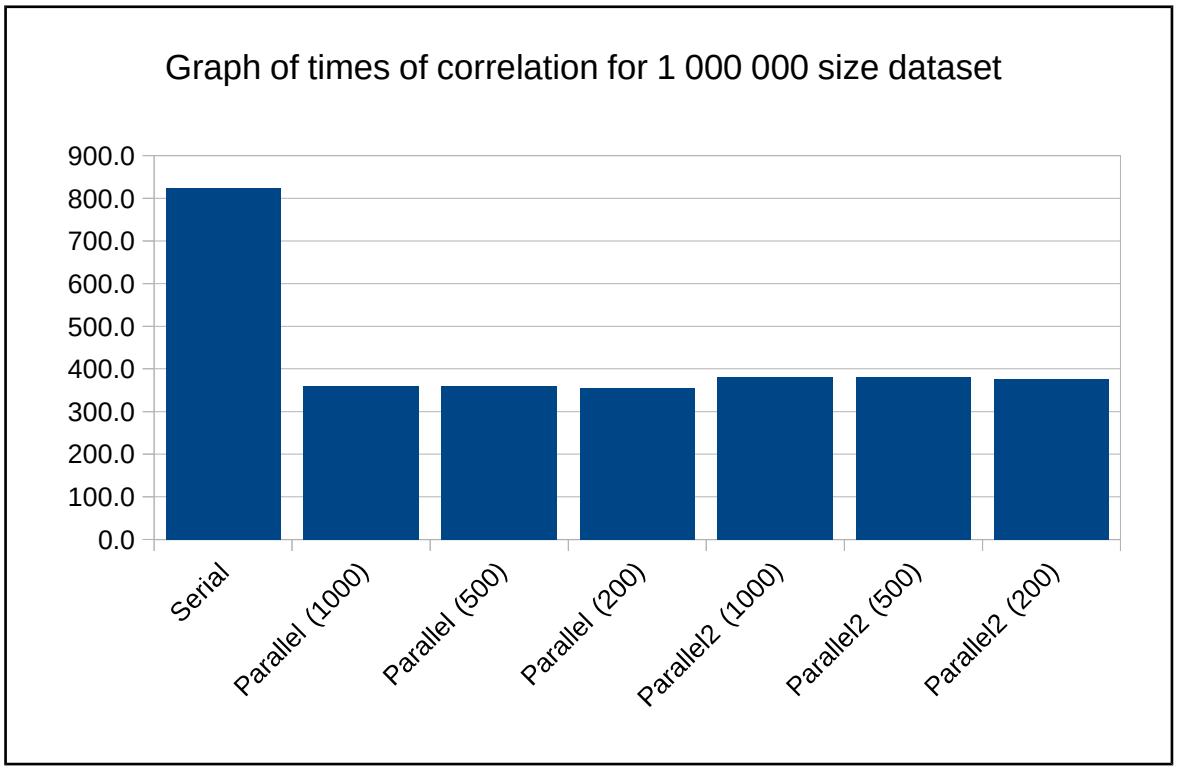
Parallel2 (500)						
	Run Time (seconds)					Average
Size of Data	1	2	3	4	5	(3 Fastest)
10000	0.088	0.046	0.043	0.043	0.041	0.042
100000	3.873	3.530	4.064	3.553	3.531	3.538
1000000	383.993	377.008	393.976	470.992	380.264	380.422

Parallel2 (200)						
	Run Time (seconds)					Average
Size of Data	1	2	3	4	5	(3 Fastest)
10000	0.114	0.039	0.043	0.040	0.045	0.041
100000	3.978	3.863	3.816	3.982	3.783	3.821
1000000	409.162	378.389	374.694	375.674	392.919	376.252

Graphs




Graph of times of correlation for 100 000 size dataset
9.0
8.0
7.0
6.0
5.0
4.0
3.0
2.0
1.0
0.0
Serial
Parallel (1000)
Parallel (500)
Parallel (200)
Parallel2 (1000)
Parallel2 (500)
Parallel2 (200)


Graph of times of correlation for 1 000 000 size dataset
900.0
800.0
700.0
600.0
500.0
400.0
300.0
200.0
100.0
0.0
Serial
Parallel (1000)
Parallel (500)
Parallel (200)
Parallel2 (1000)
Parallel2 (500)
Parallel2 (200)

Parallel vs Serial
On all of the datasets, the parallel programs ran significantly faster, becoming more significant with larger data sizes.

Parallel vs Parallel2
By the results obtained, it can be seen that there was not much of a difference for the 10 000 dataset and the 100 000 dataset. On the 1 000 000 however, there was an average improvement of about 20 seconds which is only an average of about 5% faster. This may become more significant and relevant for even larger datasets.

Sequential Cutoff
The results of varying the sequential cutoff are not completely clear. For Parallel 10 000 and 100 000, a cutoff of 1000 looks better than 500 and 200 but for 1 000 000, a cutoff of 200 looks better. For Parallel2, a cutoff of 200 looks best for 10 000, 500 looks best for 100 000 and 200 looks best again for 1 000 000. There is no clear trend for Parallel2, possibly because the cutoff is reliant on the data size. A better result could be seen with more tests.

Reduction in Time
Taking the best average times for Parallel and Parallel2 and the average times for Serial, the reduction in time for 10 000 is about 53%, and for 100 000 and 1 000 000 is about 57%.

Conclusion
The correlation algorithm and the find the maximum algorithm were well suited to be implemented in parallel and this provided a clear increase in performance over the serial implementation.