**Computer Science (CSC2002S) Assignment 1: Digital Signal Processing with Java Fork/Join Framework**

Josh di Bona (DBNJOS001)

**Report**

Radar (Radio Detection And Ranging) is an echo location system that uses a short pulse of energy to reflect off and detect by doing so, objects in the surrounding area. We are asked to perform a cross-correlation of the transmitted signal and the received signal and find the max point in the cross-correlated data which would be the closest object to the radar. In this process, the received signal is shifted in steps along the time axis. For each shift, the transmitted signal and the shifted, noisy and attenuated received signal are point-wise multiplied and then added. This is a very cost demanding process and sequential methods would cost quite a lot. We have been asked to parallelize these methods to make then less expensive.

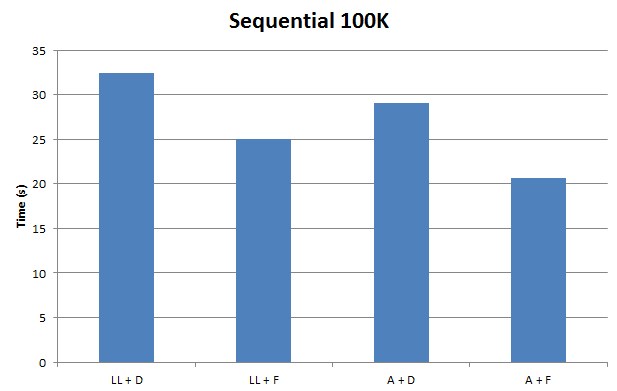
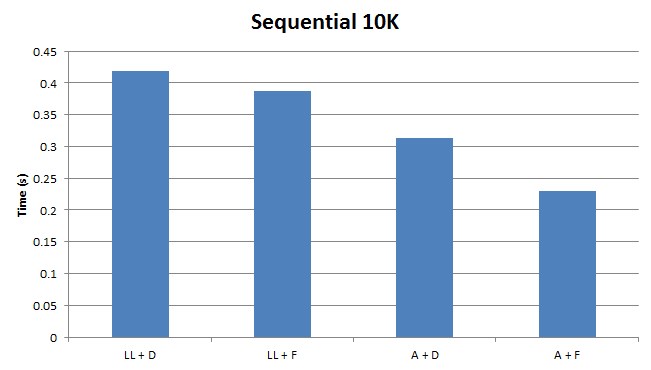
In order to see if parallelization was the best choice and to see what difference it made in the running time of the cross-correlation and finding the maximum. I first coded it sequentially and in order to get the best results I needed to optimize the sequential code to the best of my ability. So I started by testing the running time difference by using Arrays vs Linked Lists and then Doubles vs Floats.

I found that using Arrays along with Floats ran the fastest. See the times below to compute the cross-correlation and to find the maximum respectively (In seconds). The sequential cross-correlate is in Correlate.java and the sequential find max is in Max.java along with their parallel versions.

Note: Every value below is the average of 10 runs for each set. Individual run times will be given for the parallelization but not the sequential runs.

|  |  |  |
| --- | --- | --- |
| Sequential  Data Structure | Data Size: 10k | 100k |
| Linked-List + Doubles | 0.419 **|** 0.001 | 32.442 **|** 0.003 |
| Linked List + Floats | 0.387 **|** 0.001 | 25.074 **|** 0.003 |
| Array + Doubles | 0.314 **|** 0.01 | 29.09 **|** 0.02 |
| Array + Floats | 0.23 **|** 0.001 | 20.701**|** 0.001 |

Extra: It ran at 2105.05/ 2104.6569 and 0.003/0.003 for the one million size data set with the array and float combination.



For the parallelization the java Fork/Join Framework was used. It provides a very straightforward and intuitive structure to implement recursive and divide and conquer problems that can be solved concurrently. It’s a method of data distribution; the arrays were broken up into smaller work units until each unit was small enough to be worked on individually. I’ll get more into how we did that below and how we chose the correct work unit size to be worked on directly and how this affects the running time of the code.

Radar.java is the main class of the program which creates a fork/join pool and invokes the Correlate.java class and Max.java. Correlate.java contains the method compute() which like in Max, does the parallel cross-correlation and the parallel find max respectively. Each class also contains there sequential method which can also be called from Radar.java but without invoking a pool. For both find max and cross-correlate, the problem size is divided up into many small sequential segments which can be computed a lot faster.

Below are the time sets for the calculations with the sequential cut-off for cross-correlation being 500 and find max, 1000.

|  |  |  |
| --- | --- | --- |
| Data Size Used: | Time(s) to Run/Compute  Cross-correlation | Time(s) to Run/Compute  Finding the Max |
| 10K | 0.052, 0.047, 0.05, 0.039, 0.035, 0.045 | 0.001/0.0 |
| 100K | 2.27, 2.33, 2.18, 2.19,2.33 | 0.001/0.0 |
| 1M | 223.1, 205.62, 217.08 | 0.056, 0.038, 0.047 |
| 10M |  |  |

As we can see there is a massive improvement compared to the sequential code. But we still don’t know if it as efficient as it can get with the current sequential cut-off (the point where we stop the divide and conquer and calculate that portion of the array sequentially). The sequential cut-off point is important to consider because as you make it bigger you will be doing bigger sequential searches but if you make it too small you will be doing too many searches, and this could cause the running times to become not that much more smaller than the sequential running times, making parallelization ultimately pointless.

But as we can see this isn’t the case (it could be if we were working with much smaller data sets, but as you can see the bigger the data set gets the more and more beneficial the parallelization becomes). The sequential cut-off also tells your computer when to you sequential or parallel, it will always run parallel but only 1 thread will run and do it ‘sequentially’.

Below are the running times for different sized sequential cut-off points.

Cut-offs: Cross-correlate – 1000 Find Max - 5000

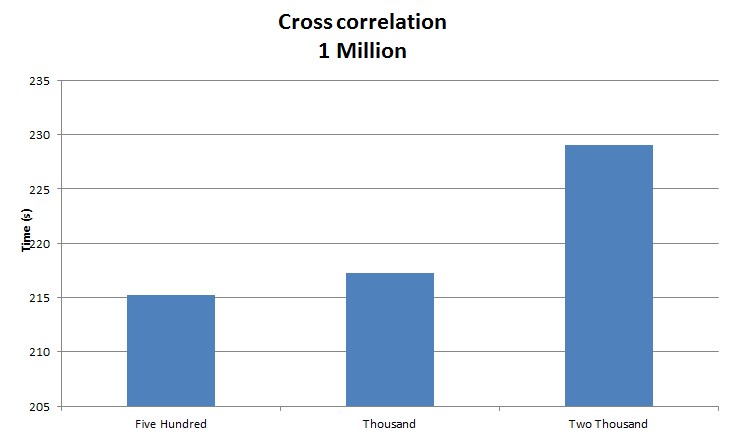
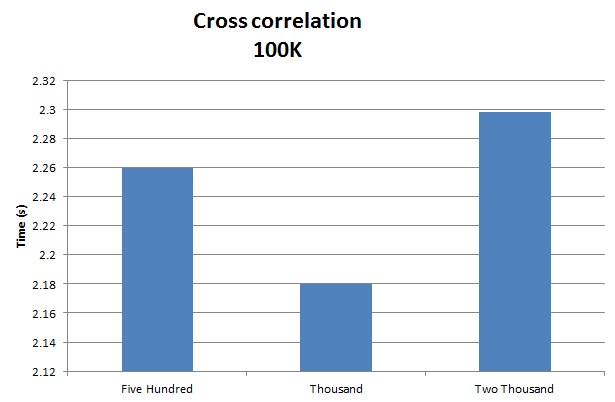
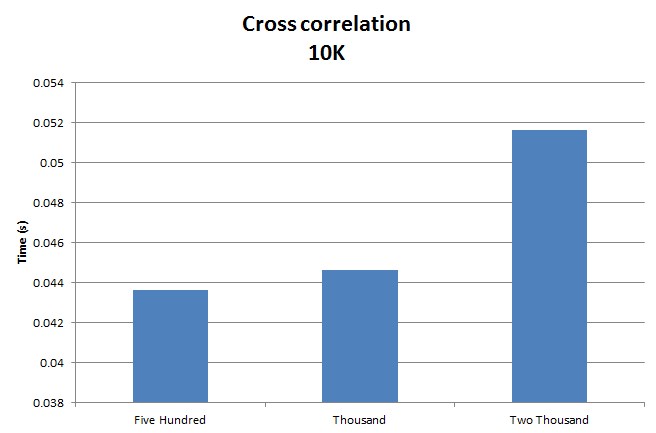
|  |  |  |
| --- | --- | --- |
| Data Size Used: | Time(s) to Run/Compute  Cross-correlation | Time(s) to Run/Compute  Finding the Max |
| 10K | 0.058, 0.036, 0.043, 0.039, 0.047 | 0.001 |
| 100K | 2.163, 2.373, 2.22, 2.298, 2.105, 2.115, 2.029, 2.14 | 0.004, 0.001, 0.0, 0.001, 0.0, 0.001, 0.001, 0.0 |
| 1M | 215.476, 214.09, 205.94, 228.8, 222.16 | 0.048, 0.039, 0.049, 0.043, 0.045 |
| 10M |  |  |

Above we can see that the cross-correlate ran a bit faster with the sequential cut-off at 1000 instead of 500, and find max with a new cut-off at 5000 instead of 1000 ran mostly the same speed, maybe a bit slower.

Cut-offs: Cross-correlate – 2000 Find Max - 2500

|  |  |  |
| --- | --- | --- |
| Data Size Used: | Time(s) to Run/Compute  Cross-correlation | Time(s) to Run/Compute  Finding the Max |
| 10K | 0.07, 0.044, 0.048, 0.048, 0.048 | 0.002 |
| 100K | 2.289, 2.397, 2.558, 2.226, 2.388, 2.252, 2.236 | 0.007, 0.006, 0.001 |
| 1M | 219.897, 226.619, 240.761 | 0.079, 0.075, 0.078 |
| 10M |  |  |

Below are the graph representations of the time differences with the different sequential cut-offs with the different sized data sets.



From the runs above we can see that increasing the sequential cut-off for the cross-correlate up to 2000, causes it to run much slower, especially as the data set size increases. With the find max we can also see quite a significant increase in running time than with 1000 or 5000 as a cut off, the balance between the number of threads created and the size set of each one wasn’t correct.

So from these tests it seems like the best cut-off points for the cross-correlation is 1000 and for find max 1000 as well. I didn’t do runs for 10 million for all of those as my personal computer wont handle it, but below is the tests for the selected cut-off points running on different hardware.

The tests above were run on a desktop computer with the following significant hardware:

Processor: Intel® Core™ i5-3570K CPU @ 3.40GHz

Installed memory (RAM): 8.00 GB DDR3 1600MHz

System type: 64-bit Operating System (Windows 7)

|  |
| --- |
|  |

|  |  |  |
| --- | --- | --- |
| Data Size Used: | Time(s) to Run/Compute  Cross-correlation | Time(s) to Run/Compute  Finding the Max |
| 10K | 0.119, 0.159, 0.09, 0.09, 0.109 | 0.009, 0.009, 0.009, 0.009 |
| 100K | 6.57, 6.448, 6.587, 6.54 | 0.009, 0.009 |
| 1M | 654.482 | 0.362 |
| 10M |  |  |

The tests above were run on a laptop with the following significant hardware:

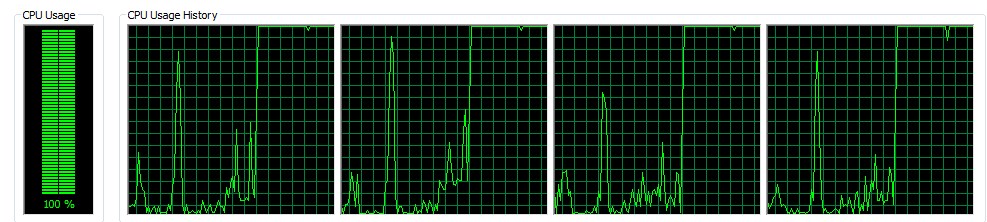
Make: Acer Aspire 5733

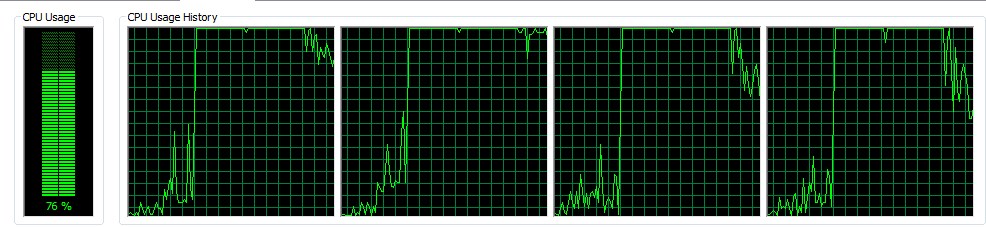
Processor: Intel® Core™ i5-580M CPU @ 2.67GHz

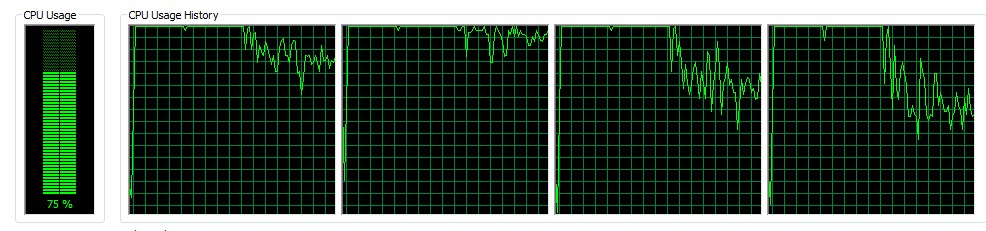
Installed memory (RAM): 6.00 GB DDR3 1333MHz

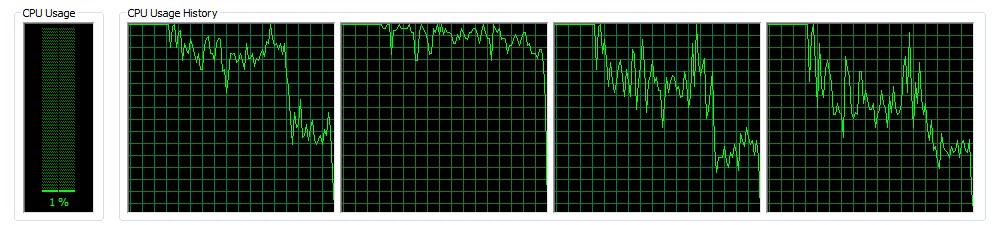
System type: 64-bit Operating System (Windows 7)

In order to check that the code was correct, I ran both the sequential and parallel methods and compared the maximum found for each. I found that the sequential method was correct when I got the same output for the 10K size file as given to us in correlatedAnsCorrected.txt.

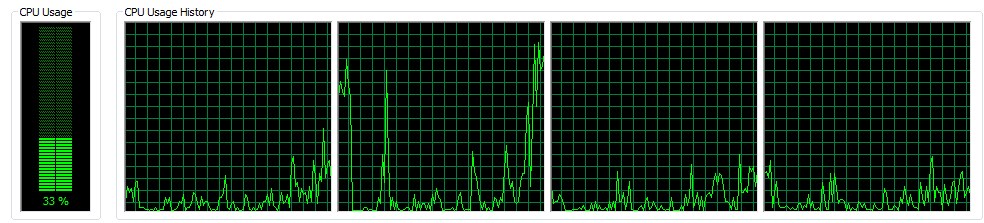
Below are the graphs drawn by the computers task manager -> performance. The below screen shots display the process of a parallel 1million cross-correlate and find max.





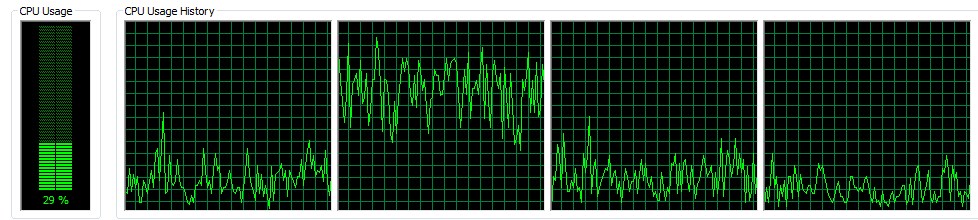


As we can see the CPU usage starts to drops as the threads all begin to finish 1 at a time.

Below is the process for a sequential compute of 1million data size for cross-correlate and find max.







In conclusion we can clearly see that the parallelization of the two methods definitely had a huge effect on the run time of the code. Above we can see how parallelization takes use of all the cores unlike sequential which can only go so fast. Although parallel might always seem like the better option it has its limits and a right balance between how big you make your sequential cut off and how many you make has to be found in order to make it worth your while to program it with multi-threading. Sometimes, especially on smaller data sets sequential methods will be better than parallel. In this case, parallel was definitely worth it especially when it came to bigger data sets making the code run about 10 times faster than it did sequentially.