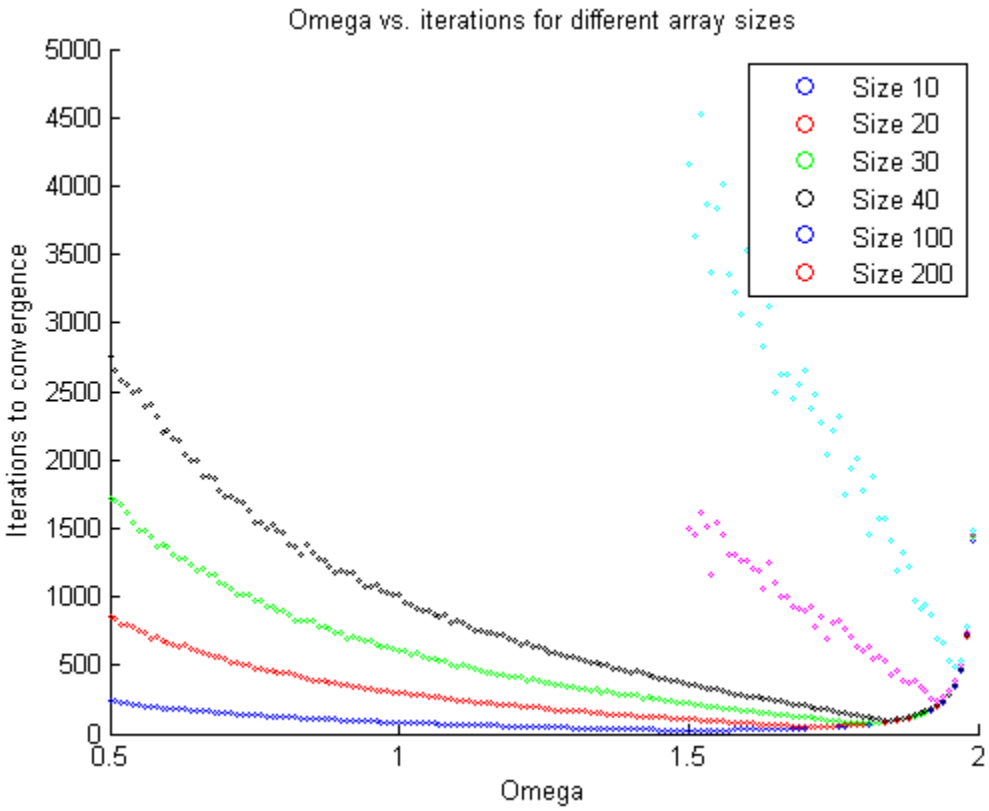
Mikhail Andreev

EC527 Assignment 2

Using hpcl-18 (2.53 GHz)

Part 1: Finding OMEGA



The overall shapes of the graphs are V-like or U-like. When Omega is small, the iterations to convergence are large, and start larger for larger array sizes. As the Omega increases, the iterations needed decrease until they reach a minimum point. After that minimum, then number of iterations increase again. The larger the array size the sharper the change from decreasing iterations to increasing iterations becomes.

From the values graphed we can see the minimum Omega, that is the Omega with the lowest iterations to convergence, is here:

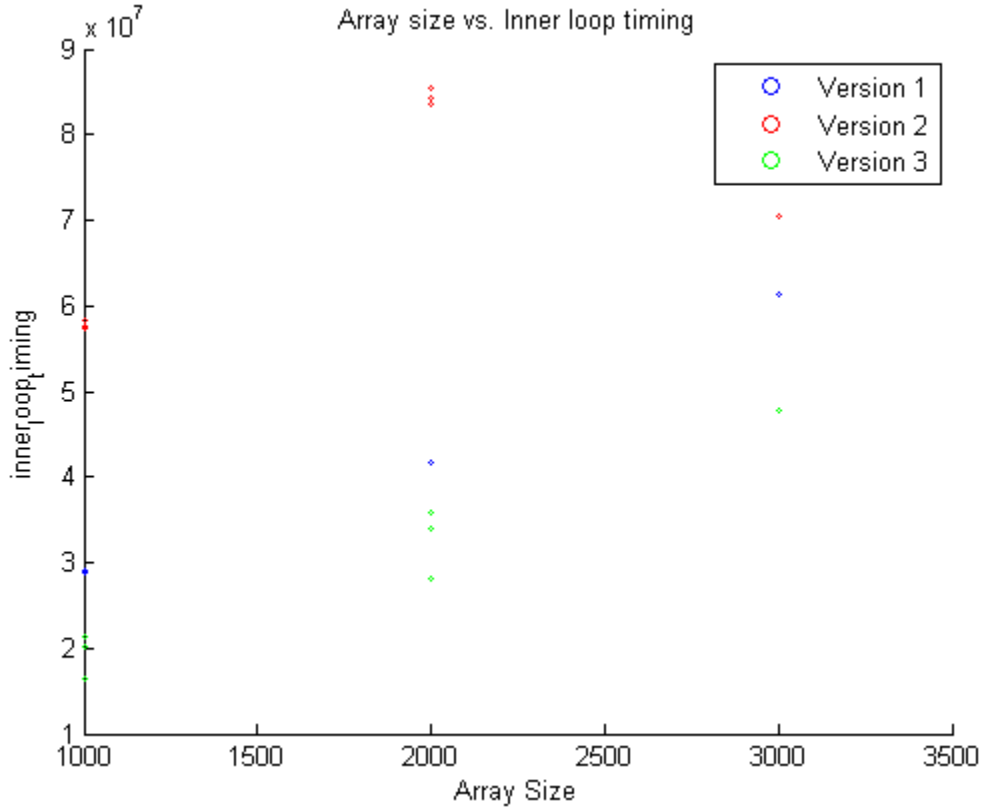
```
Minimum Omega for Size 10 is 1.500000
Minimum Omega for Size 20 is 1.710000
Minimum Omega for Size 30 is 1.800000
Minimum Omega for Size 40 is 1.840000
Minimum Omega for Size 100 is 1.930000
Minimum Omega for Size 200 is 1.960000
```

From these values, we can see that as the array size rises, so does the optimal Omega value for that array size.

Different Omega selections have different levels of sensitivity on the iterations to convergence. When the array size is small, different Omega choices have little change on the iterations to convergence. When the array size becomes larger however, the effects become much more pronounced. Sensitivity also greatly increases after the optimal value, so for larger array sizes, increasing the omega past the optimal value causes a very large spike in the iterations to college.

Assuming the Omega value is optimal, the size of the array will affect the iterations to convergence. Even when comparing iterations to convergence for different array sizes at optimal Omega, larger array sizes will require more iterations.

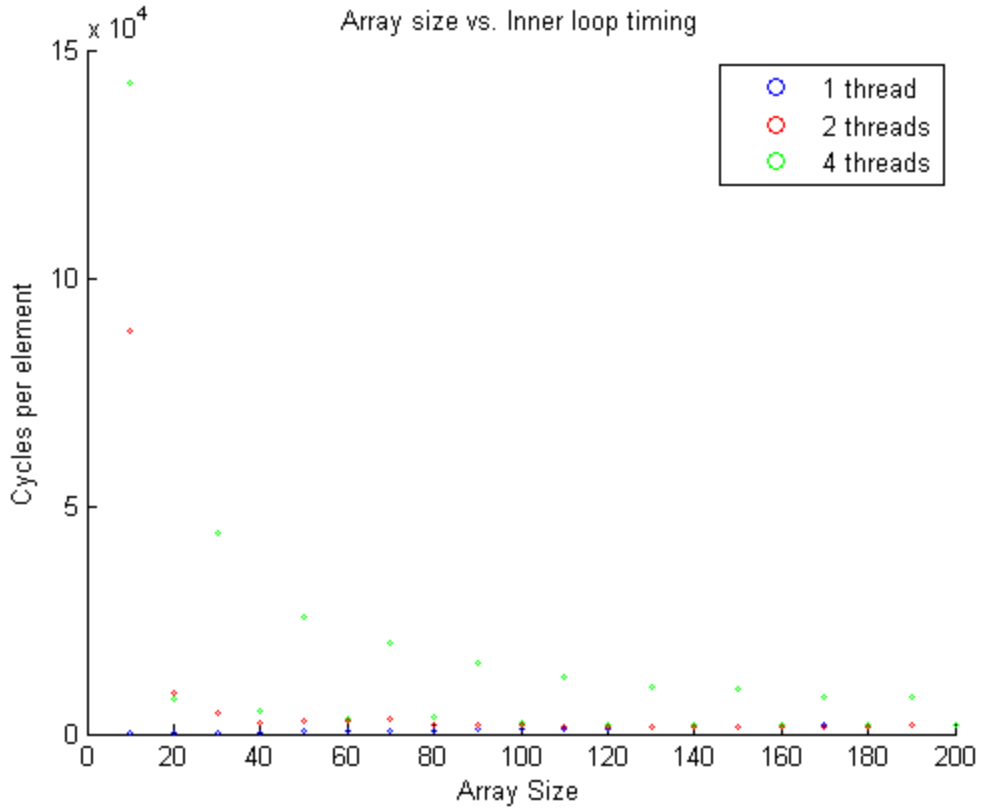
Part 2: Serial SOR Optimizations

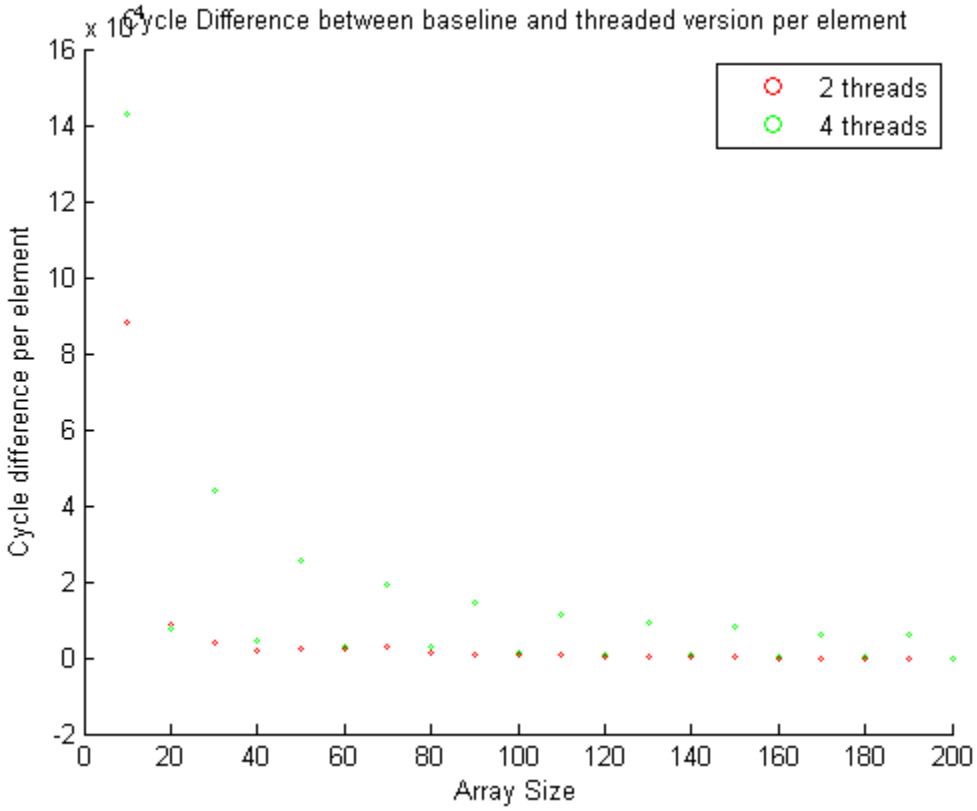


The results for the experiments are shown in the graph. Version 1 is the base version of the code with no changes. Version 2 is the index reversal, and Version 3 is the blocking version. The surprising thing about these results is that the index reversal seems to improve with larger array sizes. Although it consistently takes longer than the other two versions, which can be attributed to bad cache alignment, it seems to improve in per element runtime at higher sizes. This could be happening because the arrays become so large that cache misses become much more frequent, causing a per element decrease.

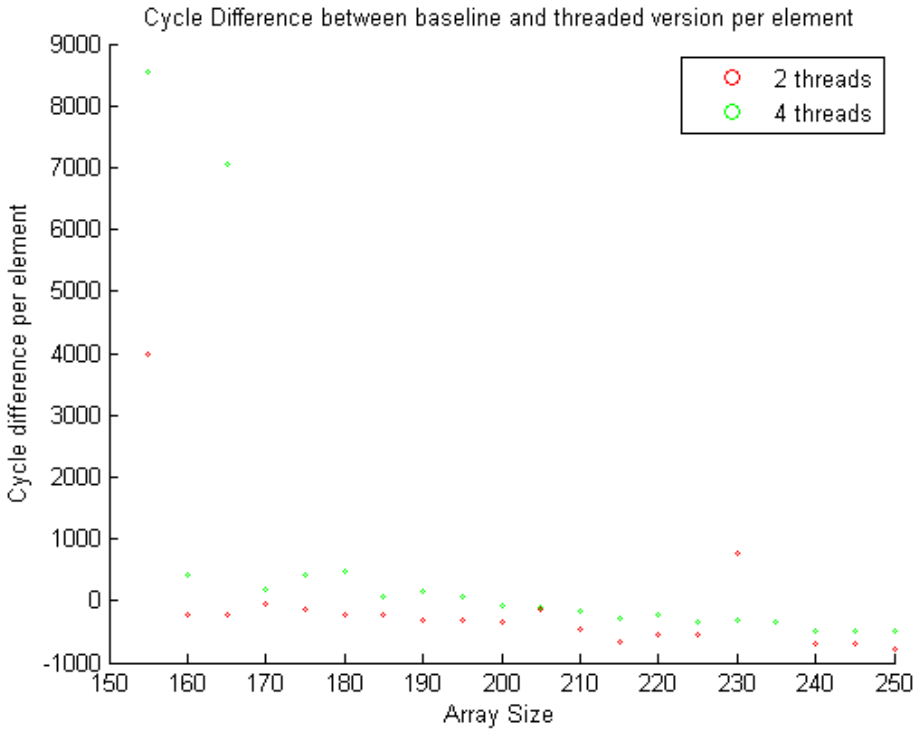
The other surprising thing is that for the blocking version, the smaller the block, the faster the performance. This is unexpected because usually larger cache blocks indicate better memory allocation. However, in this case, due to the size of the array this becomes less useful.

Part 3: Cost of Multithreading

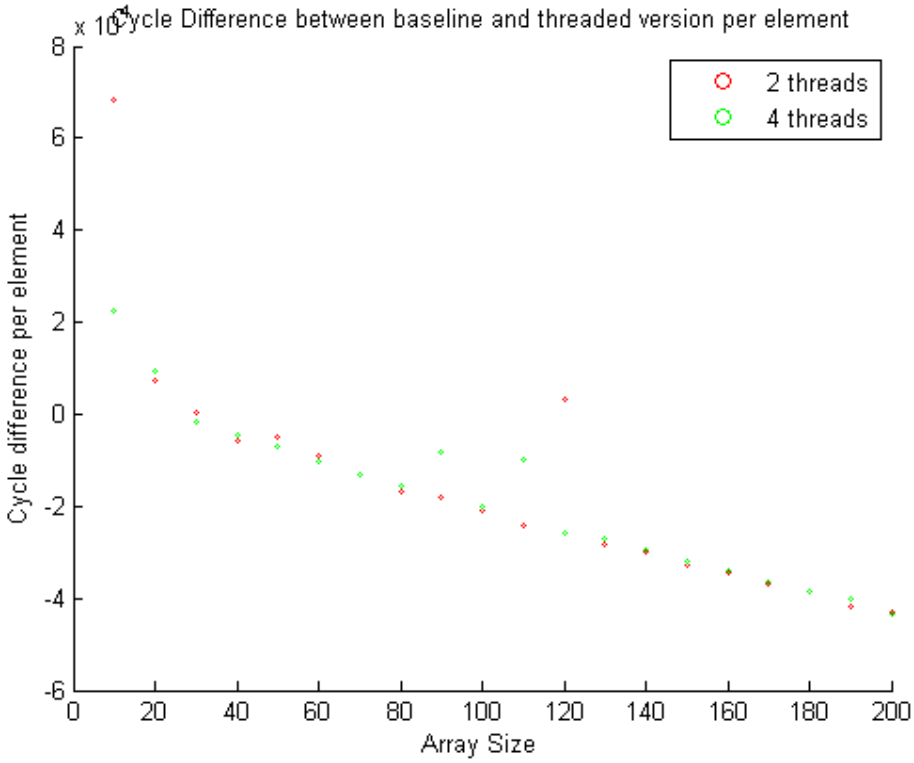




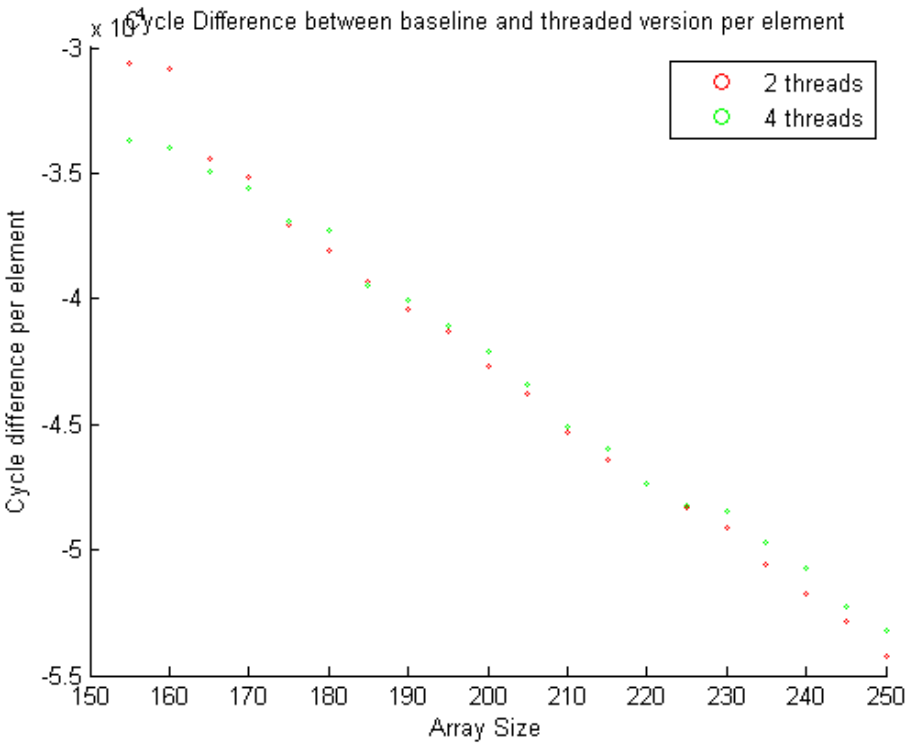
These are graphs of the performance when using the simple function. The first graph is the overall performance of the variations of different threads per element. As can easily be seen the cost of running pthreads is high initially per element, but then decreases as size goes up and efficiency increases. The second graph shows the cycle difference between the pthreaded variations and the baseline case. This is useful for calculating pthread overhead. Since we expect the overhead to be constant regardless of how much data is used in the pthread, we can focus on the first data point. Here we can see that the cycle difference between the 4 thread case and the baseline is about 145 thousand cycles. The difference between the two threaded case is about 90 thousand cycles. Since we can assume the actual computation time will be the same, the difference listed here is the overhead per array element. Since there are 10 elements, the total overhead for 4 threads would be 1.45 million cycles, and for 2 threads it would be .9 million cycles. This gives us an overhead per thread of around .4 million cycles. This may seem to be a lot initially, however, it soon becomes negligible with the calculations as can be seen in the later portions of the graph when the difference becomes equal to 0.



Cycle difference for simple calculation on array sizes [150,250]



Cycle Difference for complex calculation for array sizes [0,200]



Cycle difference for complex calculation for array sizes [150,250]

Using the above graphs we can calculate the break-even point. For the simple function the break-even point only comes around array size 205. Meanwhile, the complex function has a break-even point around 30 cycles.

Part 4: Multithreaded SOR

Code attached.