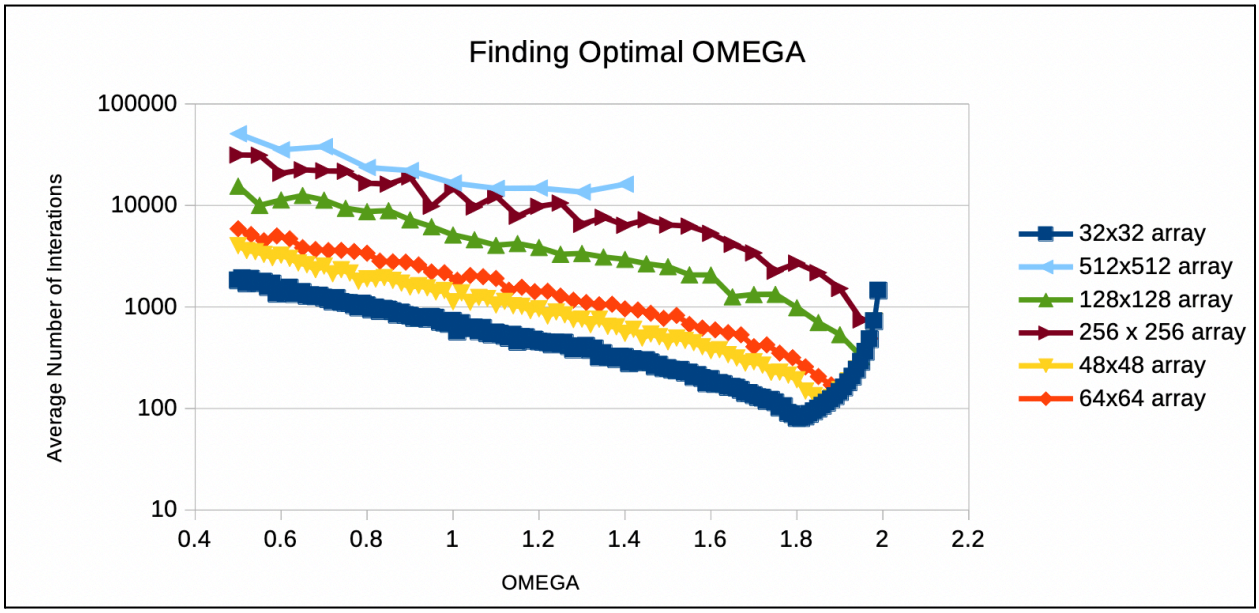
Varsha Singh
EC 527, Lab 5
Due Wednesday, March 5

Lab 5, EC527

Varsha Singh

Part 1 – Finding Optimal OMEGA

To approach this, I started with an array size of 32x32, as given. I then scaled the array size up and inversely scaled the number of tests, omega stride, and number of omega values tested. The following graph is produced:



For the data_t type being double, the largest array that will fit in a L2 cache for Intel® Core™ i7-9700 Processor (256 KB), is an array size of about 181 x 181. To get values above this array size, I calculated the values for a 256 x 256 and 512 x 512 sized array. It seems like the overall shape of the OMEGA curve will be a downwards trajectory, except for the 32 x 32 smallest array size that seemed to spike up in the tests with an OMEGA value of over 1.8. It seems that the optimal omega differs for each array size.

Optimal OMEGA values:
32 x 32: 1.81
48 x 48: 1.86
64 x 64: 1.88
128 x 128: 1.95
256 x 256: 1.95
512 x 512: 1.3 (did not check for the entire range)

Based on the trend, it seems like the values for OMEGA seem to increase as the array size gets larger. I predict that the same behavior would be observed with the 512 x 512 array, but the time it took to run was too long to reasonably test.

Varsha Singh
EC 527, Lab 5

The sensitivity of the OMEGA selection seems to be larger for the larger matrices as well. Compared to the values for OMEGA of the 32 x 32 matrix with a stride of 5, the difference between values is much smaller than for the 512 x 512 value. For example, For a 32 x 32 array with the optimal omega, 1.81, the average iterations was 80.2. For an omega of 1.86, the average iterations was 106.1. For a 512 x 512 array with the optimal omega, 1.3, the average iterations was 13553.8. For an omega of 0.8, the average iterations was 23641.8. The scaled difference is much greater in this sense.

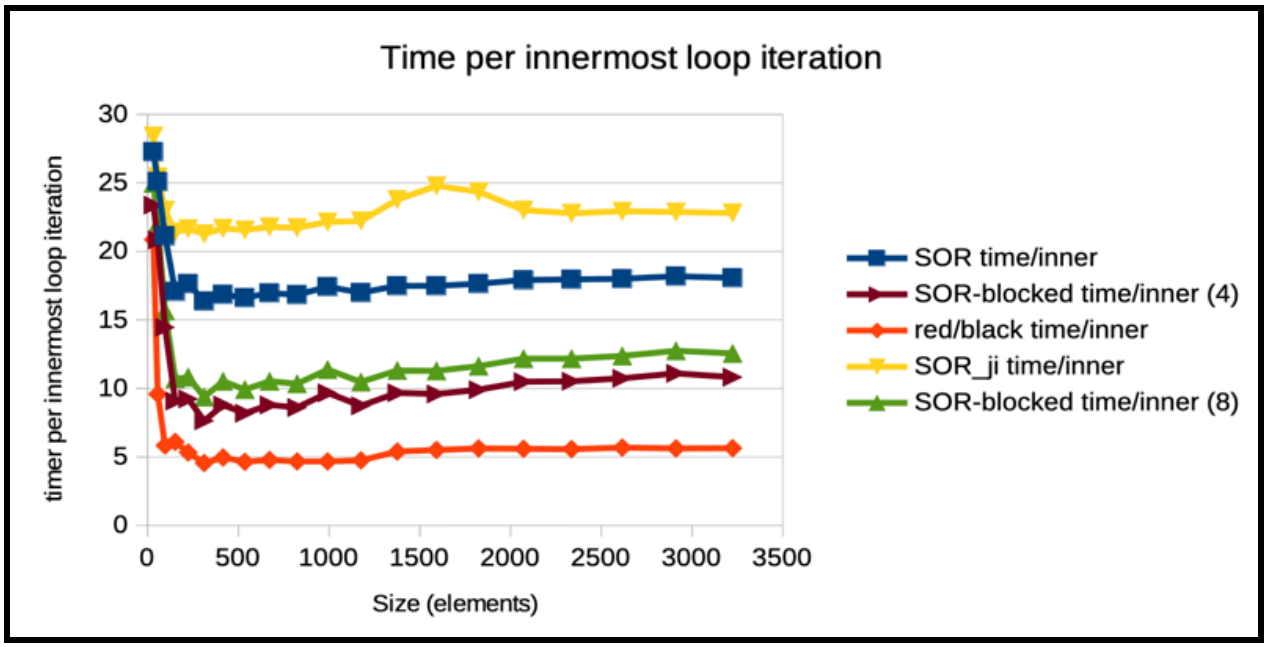
Near the optimal OMEGA, it seems like what affects the number of iterations to converge has to be the grid size (size of the array). A larger array has more unknowns which will usually lead to more variables that can change before convergence. This would back up the data, as it takes a larger array much longer to converge on average as compared to a smaller array, even with the same omega values.

Part 2 – Serial SOR optimizations

I found the time per innermost loop iteration by multiplying the number of iterations it took to converge by the size of the matrix. I found that the iteration for SOR was the same for standard, ji, and blocked, but that the time differed. Also, red/black was much faster and took less iterations for the smaller matrices, but much more (~4x) for the largest matrix size: 3224 x 3224.

I am surprised by the results. I thought that ji would perform better that ij from our previous labs, but this experiment shows that it is on average the worst performing SOR method. Additionally, I would think that based on previous labs (MMM) that larger blocks that fit in L1 cache would have a better performance, but it seems here that a block size of 4 performs better on average than a block size of 8. I think the best explanation for this is that SOR does not work for matrices in the way that matrix-matrix multiplication does. For matrix-matrix multiplication, it makes sense to calculate for a block at a time because if the array size is small enough, then you will have a higher cache hit rate. For SOR, it is best if we are able to take the entire array and work with the neighbors of the current data point to relax the current index. Thus, a smaller block would spend less time fetching the block because there is less to fetch, and it will still be around the same for cache hits per iteration. Surprisingly, red/black was by far the best performer and maintained as the most efficient method for each array size from 32 x 32 to 3224 x 3224.

Varsha Singh
EC 527, Lab 5



Part 3 – Cost of multithreading

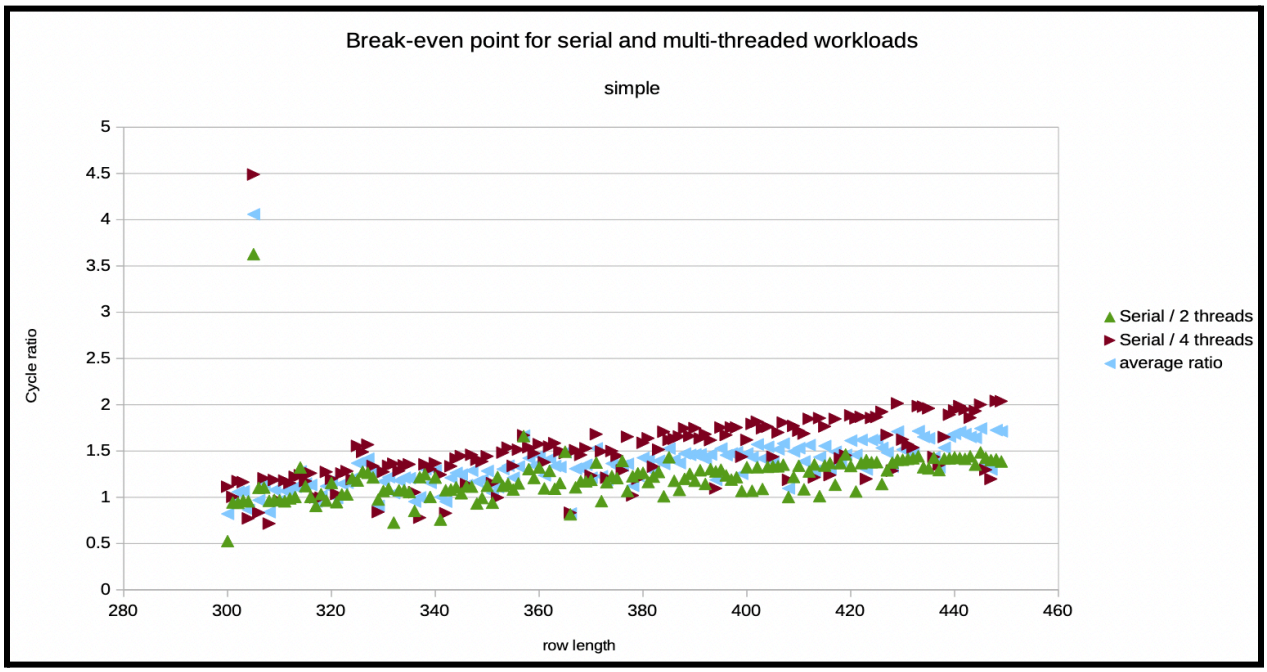
I used an array with row_len being 7 and calculated values of row_len between 1 and 200. With this, I found that the cycle ratio of the four-threaded version to two-threaded version was 2.23715434161461. This is the closest ratio to a ratio of 2, so it will be the approximation for estimating the cost of multithreading. I divided the difference between the two versions' cycle count and found that the cost per thread is about 229.914 microseconds.

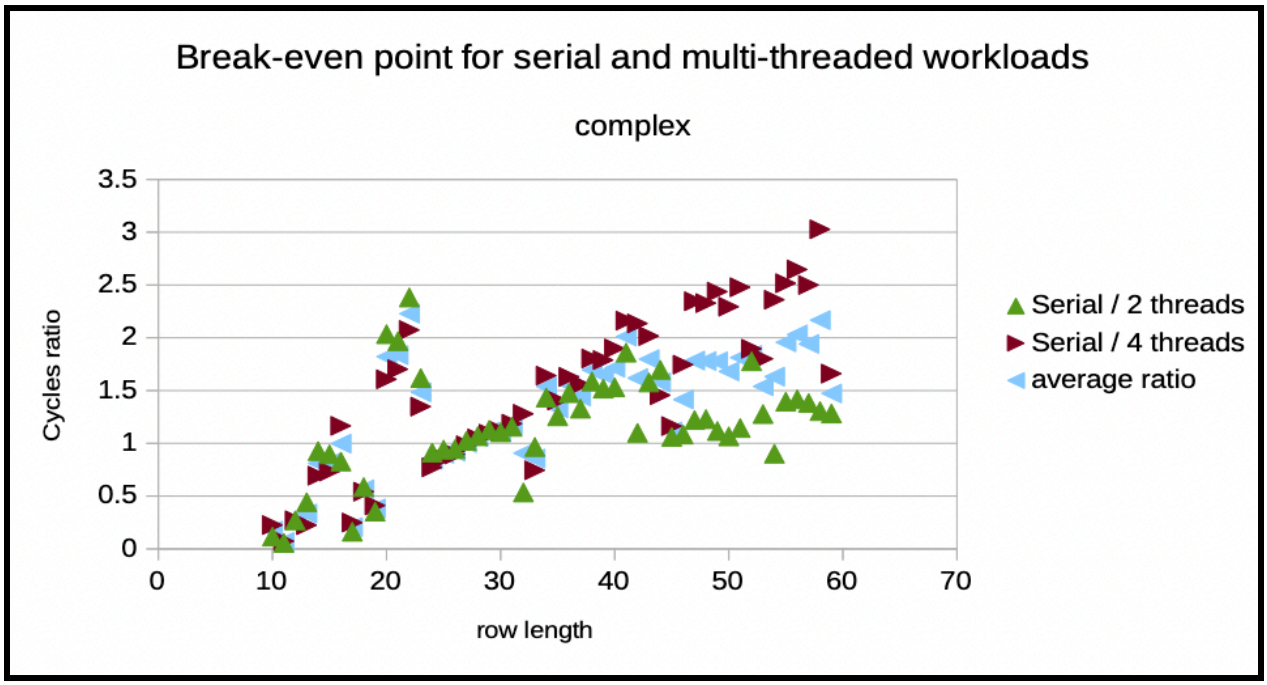
When running the complex function, I found that the tradeoff between the serial version and multi-threaded version was around an array size of 27 x 27. From the array sizes smaller than this, generally the average ratio between the serial version and two multithreaded versions is smaller than 1, whereas the average ratio between the serial version and two multithreaded versions with array sizes larger than this is generally greater than 1. The average ratio at this array size is 1.00289004215591.

When running the simple function, I found that the tradeoff between the serial version and multi-threaded version was around an array size of 337 x 337. From the array sizes smaller than this, generally the average ratio between the serial version and two multithreaded versions is flippant, with some values being a little greater than 1 and some being a little below 1, whereas the average ratio between the serial version and two multithreaded versions with array sizes larger than this is generally greater than 1. The average ratio at this array size is 0.998567494346623.

It makes sense that a more simple function will require a larger array to find the tradeoff between time to setup and join the thread as compared to a more complex function. What I find

interesting is that the number of cycles required for the trade-off point between the simple and complex methods differ. For the simple method, the serial version uses 526182 cycles, whereas for the complex method, the serial version uses 136581 cycles. It requires a lot more cycles for a serial version to be worse than a threaded version for a simple method, as compared to a complex method.



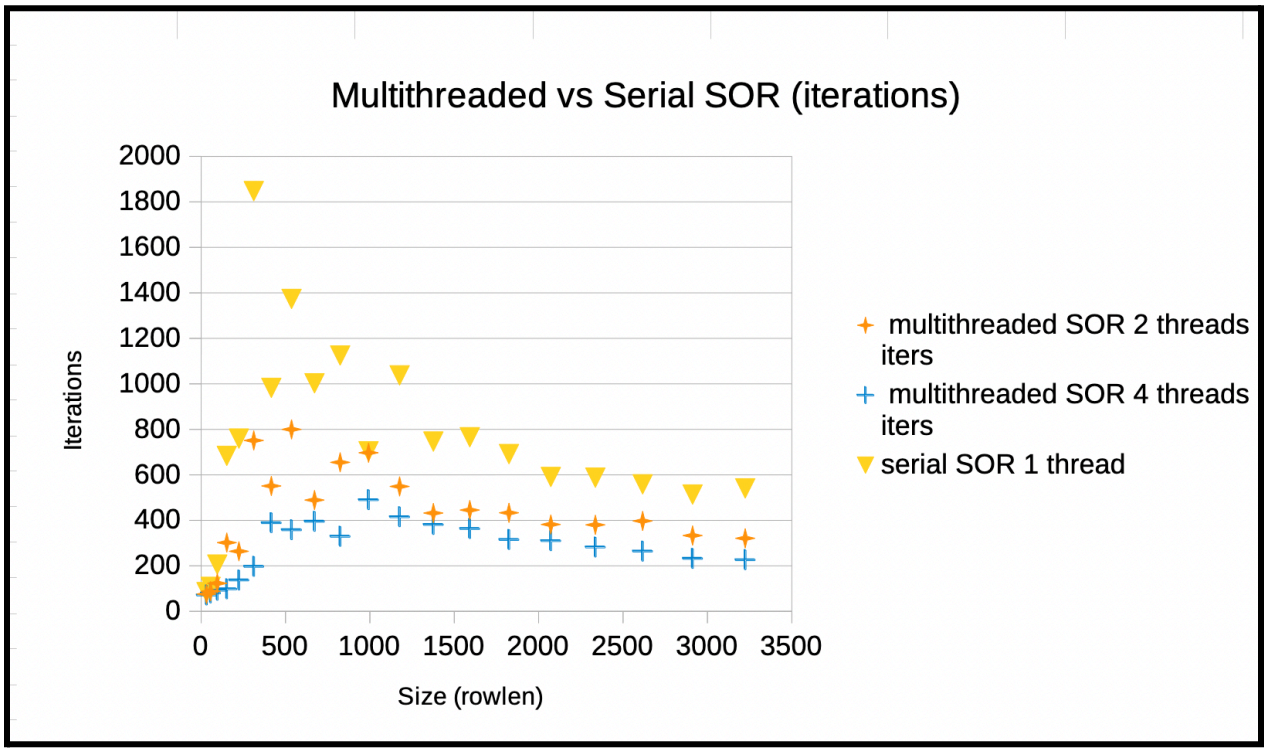


Part 4 – Multithreaded SOR

As per the new instructions, I attempted to develop a multithreaded SOR version that branches off of the serial SOR implementation that was given. I set my OMEGA value to 1.85 because it

appeared to be an overall optimized value as per my experiment from Part 1. I kept the same script that tested for array sizes using the quadratic function, and found that my results might not be as expected. Because SOR relaxes values at a timestep, I expected that having a multithreaded program might converge faster if not around the same number of iterations as previous cases because the logic implemented should block threads from continuing at a future timestep before all threads are completed. Because I was having trouble with my code at first, I believe I used more barriers and a mutex than might be needed. Because of this, the multithreaded version takes longer as compared to the serial version. This comes from threads waiting for other threads to reach the barrier. This could additionally be caused due to the time taken for context-switching between threads, but I believe that my results from part 3 show that this time is negligible for most of the array sizes tested. However, I assert that my use of mutex and barriers adds considerably to runtime because I tested a version that did not include the barrier or mutex and it operated at a much faster real-time as compared to the real-time of the serial SOR. This is expected because of the logic utilized in SOR. If some threads are slower than others, it will use a more converged value at a future timestep to converge its own values, which will still be forward progress and should not raise the expected number of iterations until convergence. I tested my multithreaded version with 2 threads and 4 threads, and compared both the number of cycles and number of iterations to the serial SOR given.

I expected that the number of cycles should remain almost the same as the serial version, but the number of cycles should have decreased. With a more bold implementation that does not conservatively employ barriers, this might have been more apparent.



Varsha Singh
EC 527, Lab 5