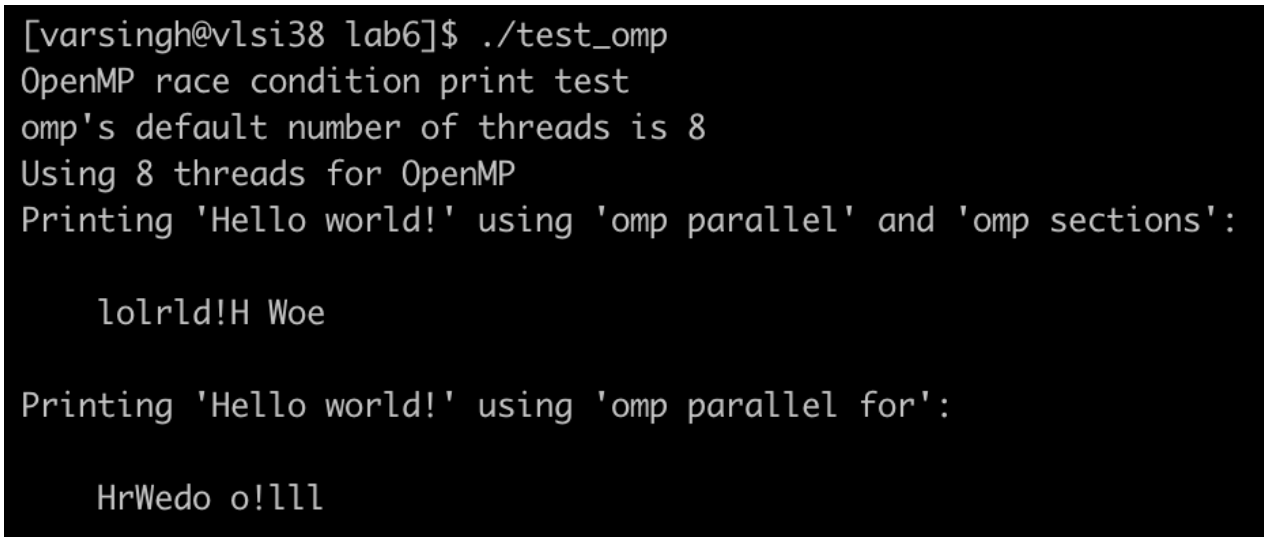
Varsha Singh
Lab 6, EC527
Martin Herbordt

Lab 6: EC527

Varsha Singh

Part 1: OpenMP basics

1a. "Hello World!" (or "ell HWoo!dlr")



```
[varsingh@vlsi38 lab6]$ ./test_omp
OpenMP race condition print test
omp's default number of threads is 8
Using 8 threads for OpenMP
Printing 'Hello world!' using 'omp parallel' and 'omp sections':

    lolrld!H Woe

Printing 'Hello world!' using 'omp parallel for':

    HrWedo o!lll
```

1b. Parallel For

I tested the given code using the given parameters, and then used this to zoom into a section to try and find the intrinsic OpenMP overhead. I did this similarly to how I approached it in the pthreads lab, and did a division in runtime between the serial code and the 4 OMP threads version, trying to find when their runtimes became about equal.

It appears that CPU-bound has a lower break even point, with rowlen being 5. The memory-bound and overhead-bound had about the same break-even point, with rowlen being 54-56 and 53-54 respectively.

Following the same fashion as the pthread lab, I found the ratio of timing of 4 threads and 2 threads, and found the 'best overhead' by finding the minimum difference from 2. I divided the average of the 4 threads / 2 threads runtime by 2 to find the average overhead per thread: 0.478688281156739. Compared to when this was done in the pthread lab: 0.000012773, both measured in seconds.

This tells me that OpenMP threads have a worse overhead than pthreads, by 0.478675508156739 seconds.

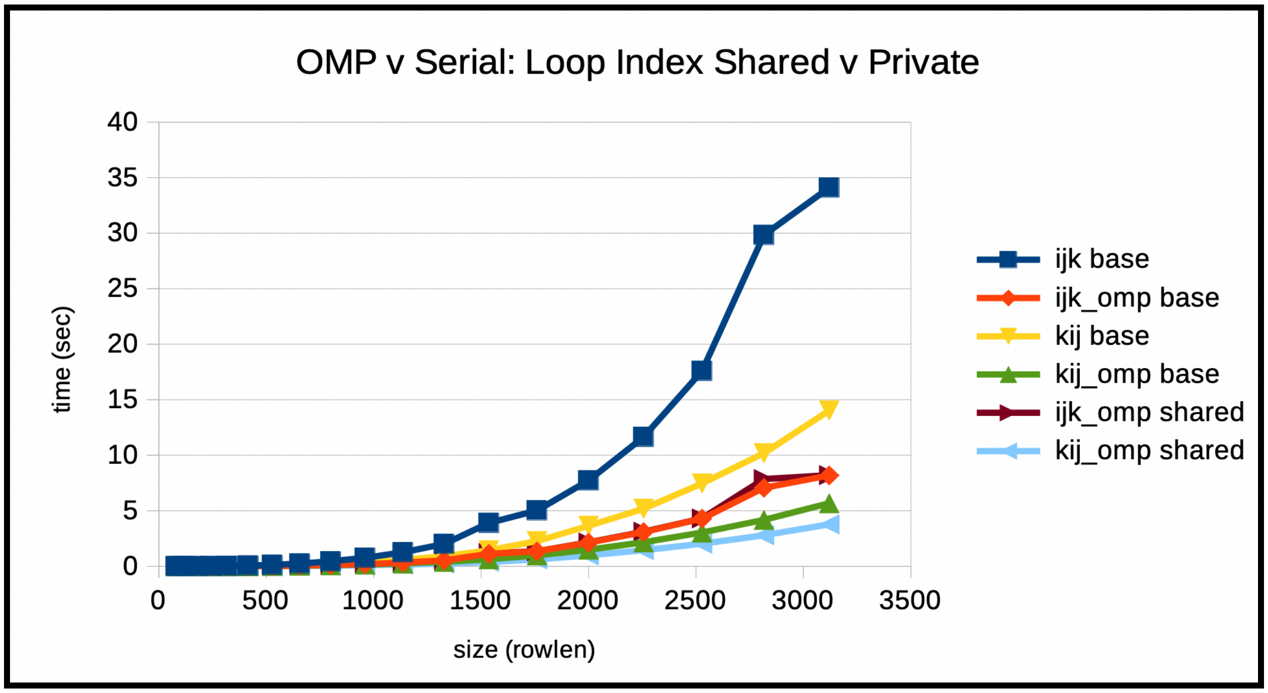
I expected that the OpenMP overhead would be greater than the overhead for pthreads. AAs explained in the assignment, OpenMP threads use barriers and pass variables to the thread in a way that was not done with the previous labs measurements, as we instead found the overhead of creating a thread, which has a finer granularity than with OpenMP.

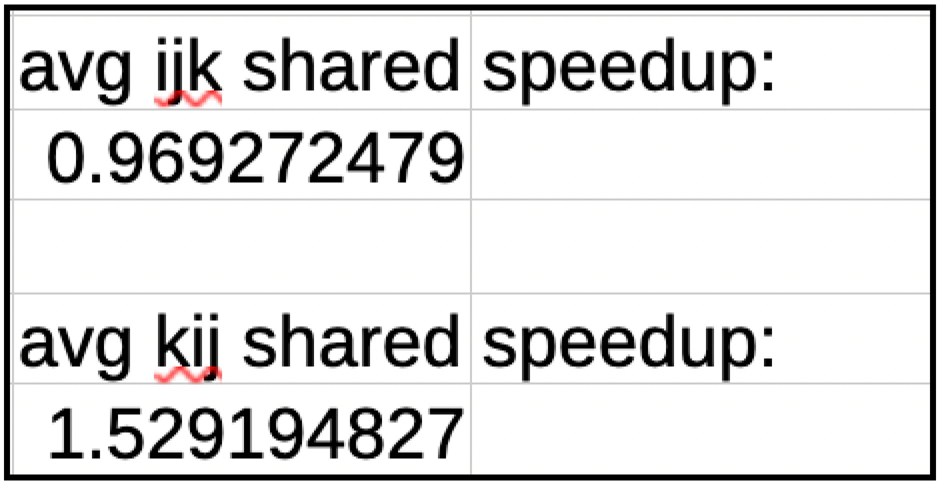
1c. MMM, 2 loop versions (serial vs OpenMP)

For this section, I edited the pragma statement to move the loop indices from the private list into the shared list. After this, I moved the loop indices back to the private list, and tried a run once where the statement was before the middle for loop and another where the statement was before the inner for loop.

For the loop index being switched between the shared and private lists:

For the base measurements, I found that ijk performed slower than kij indexing. This is the same case for the omp versions, with both omp being better than the serial versions. By switching the loop indices to shared, I found that the behavior seems to vary for ijk and kij. Because the plotted lines are so close together, it was hard to tell if there was an average gain or loss from switching the indices to shared. To evaluate this, I found the average percent speedup across all array sizes from the base omp version and the modified one with list indices in shared. I found that the average ijk shared speedup was 0.969x, whereas the average kij shared speedup was 1.529x. This indicated that ijk performed worse with the loop indices in the shared list as compared to kij. I believe that this might have to do with the overall cache efficiency of ijk being fairly poor, as compared to kij. Additionally, this might be due to false cache line sharing, which could add significant overhead and destroy the benefit of parallelism.

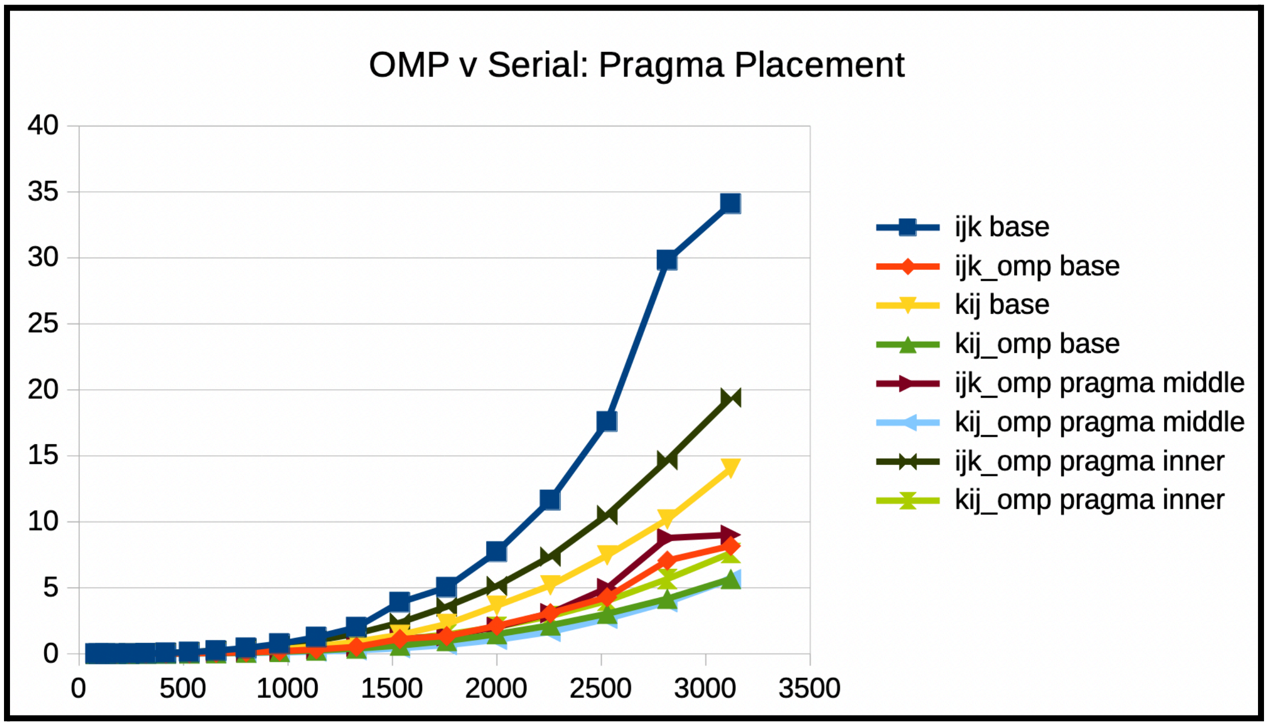


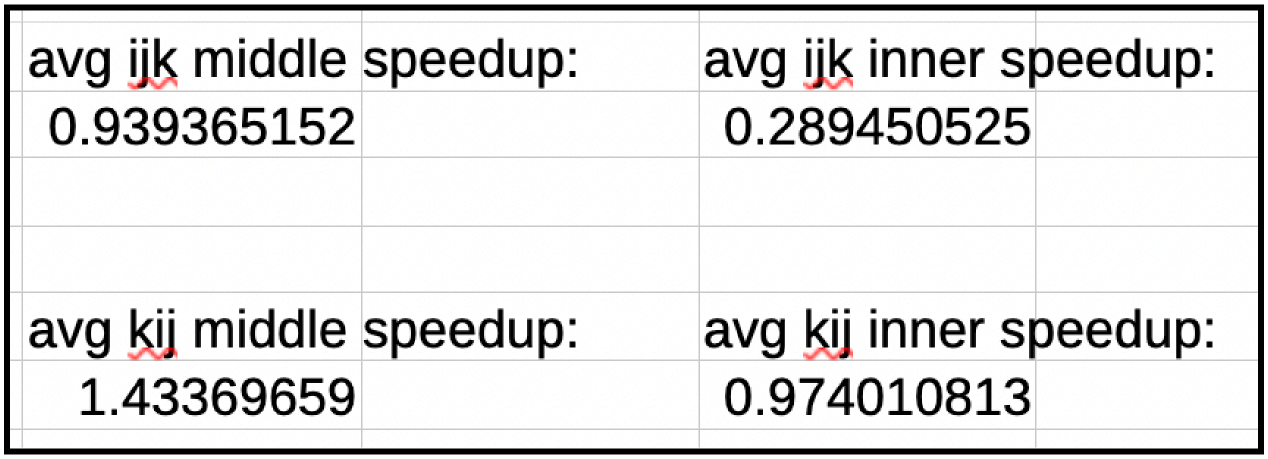


avg ijk shared	speedup:
0.969272479	
avg kij shared	speedup:
1.529194827	

For the pragma statement being switched from the outer for loop to the middle / inner loop:

I found that in general, ijk changes decreased the efficiency of the program. Like before, I found the average speedup from base to either middle or inner loop: IJK middle speedup was 0.939x, and inner speedup was 0.289x. For KIJ, I found the middle speedup to be 1.434x, and inner speedup to be 0.974x. I understand this to be due to the overhead for creating and synchronizing threads. When we put the pragma statement in the inner loop, it executes the most number of times and it appears that placing it in the middle is still executed fewer times to allow there to be a benefit as compared to an outer loop.





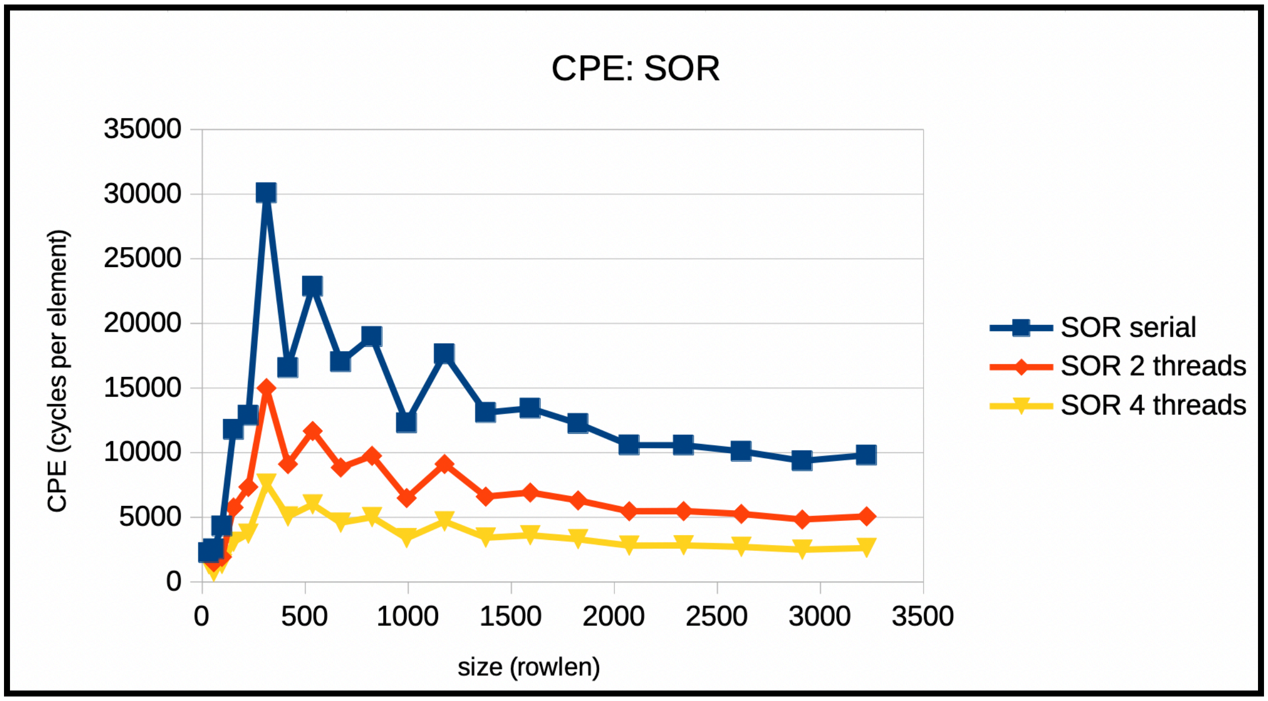
avg ijk middle speedup:	avg ijk inner speedup:
0.939365152	0.289450525
avg kij middle speedup:	avg kij inner speedup:
1.43369659	0.974010813

Part 2: OpenMP on real programs

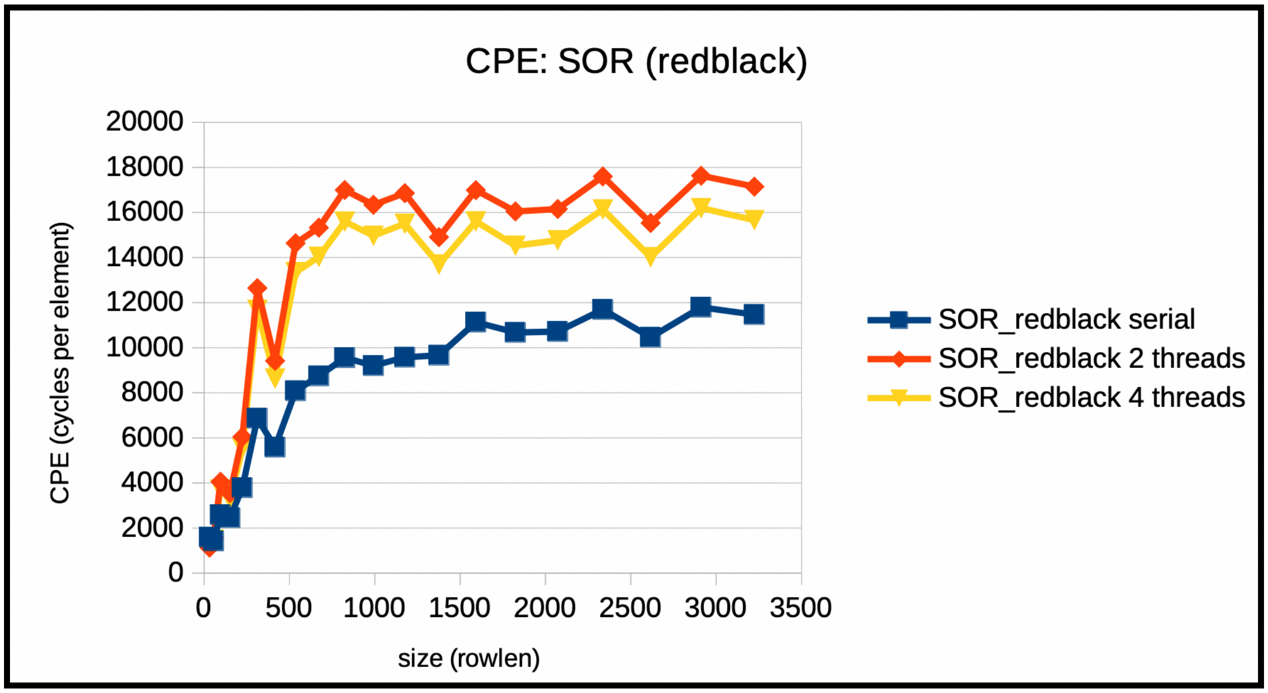
2a. SOR

I parallelized all four methods of SOR (serial, serial with indices reversed, redblack, and blocked) with OpenMP by using #pragma omp parallel for private(i, j, change, ii, jj) reduction(+:total_change) on the outer for loop for each function. I opted to do so because of the results from the previous part, in which we investigated the placement of the #pragma omp for placement. I decided to keep the loop indices private, as well as the local change that each thread updates. However, the total change is used by all threads.

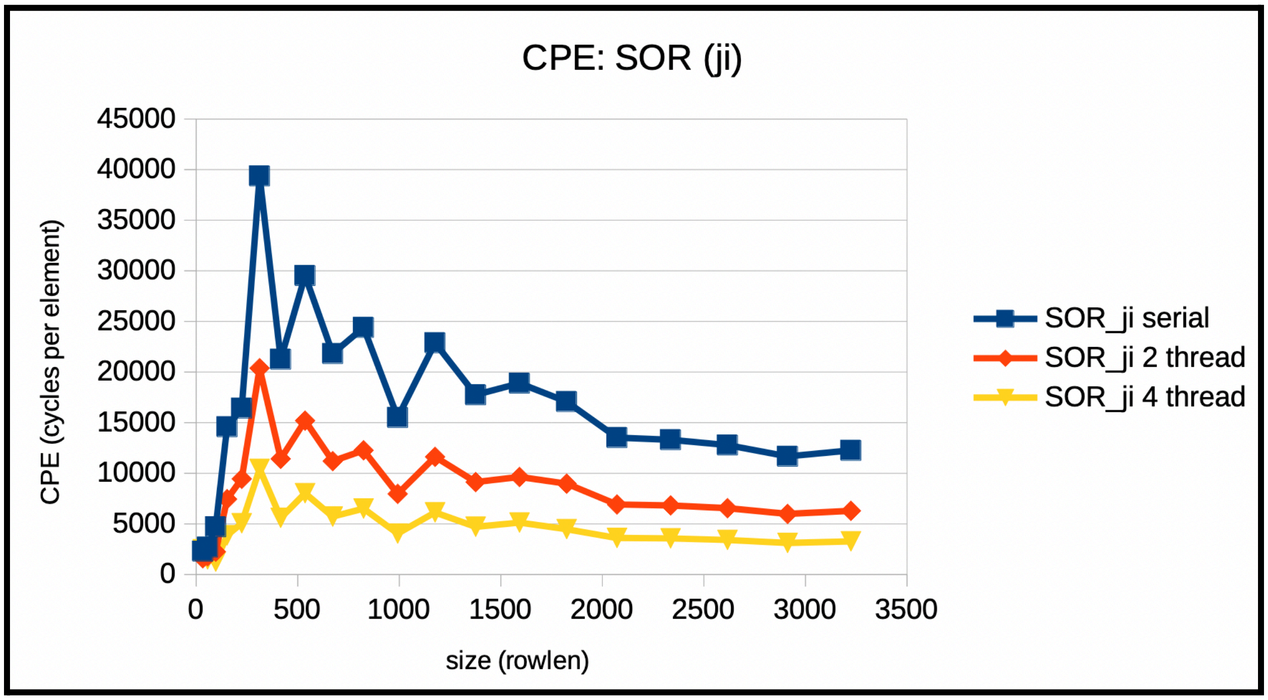
SOR had roughly a 1.9x speedup for 2 threads, and a 3.7x speedup for 4 threads. I found this by summating all times taken for the three different thread values. This tells me that the overhead for openMP with SOR is significantly lower than the benefits of parallelization.



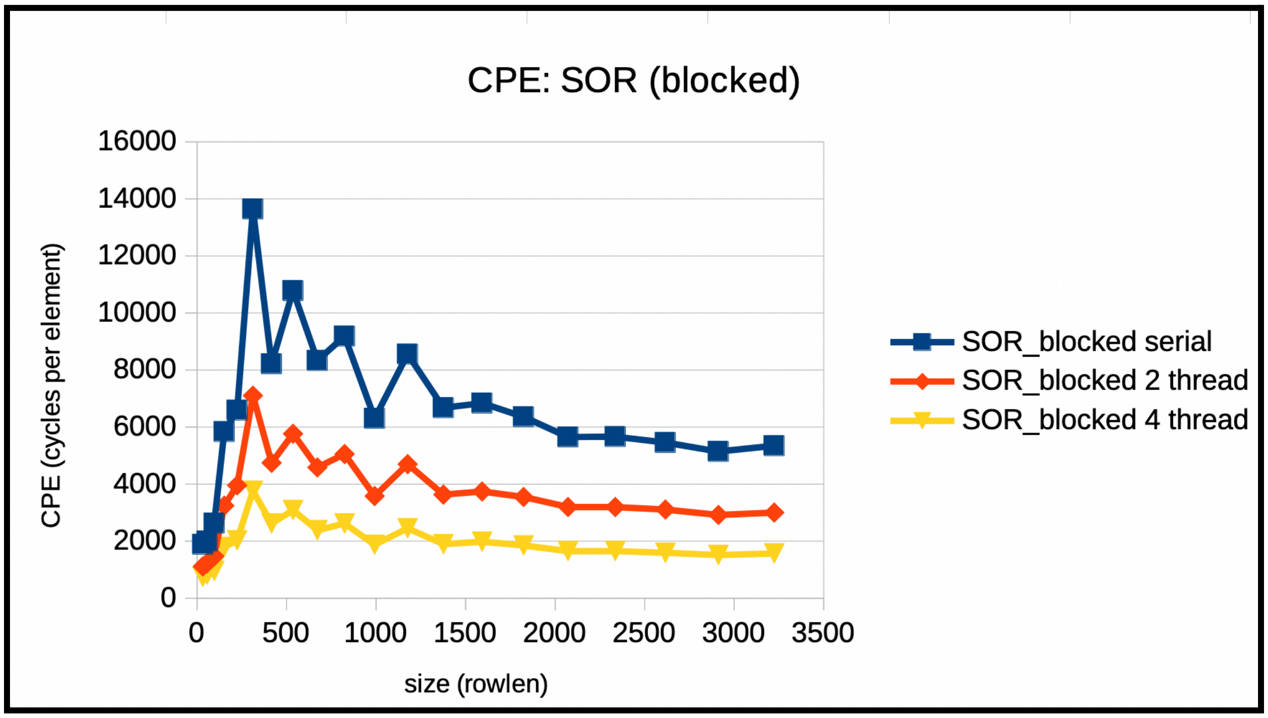
Redblack SOR had roughly a 0.6x speedup for 2 threads, and a 0.7x speedup for 4 threads. This means that the overhead with OpenMP + synchronization required is not beneficial for runtime. This makes sense, as redblack proves to be fast but does not honor the SOR expected physics, because it relaxes indices that are not adjacent to each other in a checkerboard pattern. The overhead of synchronization makes 2 thread performance and 4 thread performance poor compared to serial performance. For the smallest matrix size tested, the serial version performed worst. However, this does not remain the case for long.



SOR_ji had roughly a 1.9x speedup for 2 threads, and a 3.6x speedup for 4 threads. This tells me that the overhead for openMP with SOR_ji is significantly lower than the benefits of parallelization. This can also be said for the synchronization time needed, as it still performs better than serial significantly for larger sized matrices. For the smallest matrix size tested, the overhead for 4 threads made the CPE worse than the serial version.



SOR with blocking had roughly a 1.8x speedup for 2 threads, and a 3.4x speedup for 4 threads. I found this by summating all times taken for the three different thread values. This tells me that the overhead for openMP with SOR with blocking is significantly lower than the benefits of parallelization. This can also be said about the overhead created from synchronization of multithreaded programs. Even for the smallest size tested, the serial version performed the poorest.



2b. MMM

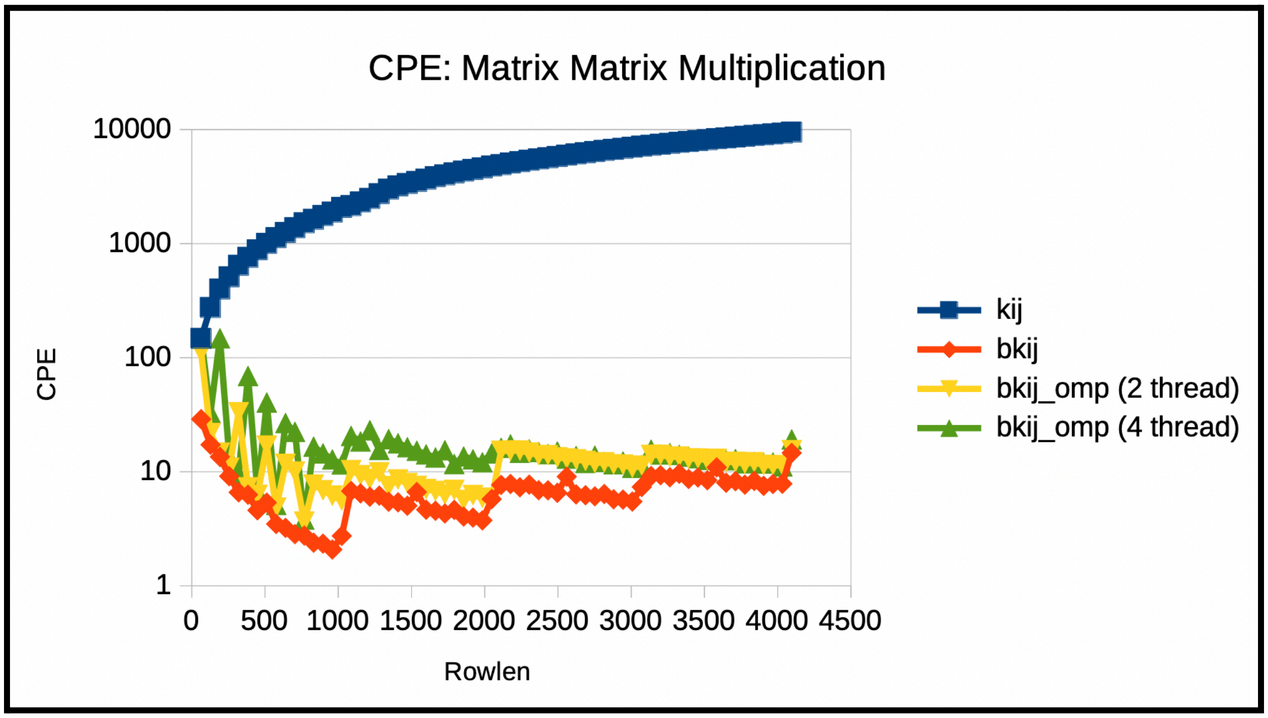
For my best matrix matrix multiplication, it was a blocked kij version that I had set the number of blocks to be 32. For this version, I put a #pragma omp parallel for private(jj, ii, k, i, j) shared(a0, b0, c0, length) statement outside of the outer for loop. I decided to keep all loop indices as private for each thread, but to keep that they can access a0, b0, c0, and length in shared. I did not declare anything for sum, as this is a private variable for each thread in execution and declared in an inner for loop.

I compared these results to a serial kij MMM version, as well as a serial blocked KIJ version, and tested for 2 and 4 threads.

For the two thread version, I found that there was an overall speedup as compared to the serial kij version of 238.8x. This can be expected, as we saw that blocking had an overwhelmingly positive effect on MMM. However, as compared to the base blocking version, we see a speedup of 0.35x. This can be attributed to the overhead of synchronization from having two threads.

For the four thread version, I found that there was an overall speedup as compared to the serial kij version of 373.3x. This can be expected, as we saw that blocking had an overwhelmingly positive effect on MMM. However, as compared to the base blocking version, we see a speedup of 0.55x. This can be attributed to the overhead of synchronization from having four threads.

Overall, this would indicate to me that parallelism can have great performance improvement as compared to serial first-pass code. However, with an effort to thoroughly optimize, the overhead for creating new threads and synchronization may outweigh the benefits. It definitely took longer to write the optimized code by hand, as compared to using OpenMP.


CPE: Matrix Matrix Multiplication
CPE
Rowlen
kij
bkij
bkij_omp (2 thread)
bkij_omp (4 thread)