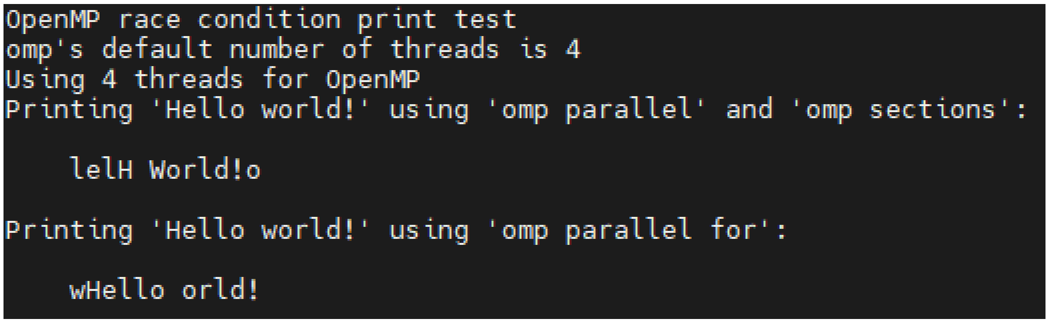
Evan Lang
Yang Lu

Lab 6 Report

1. .

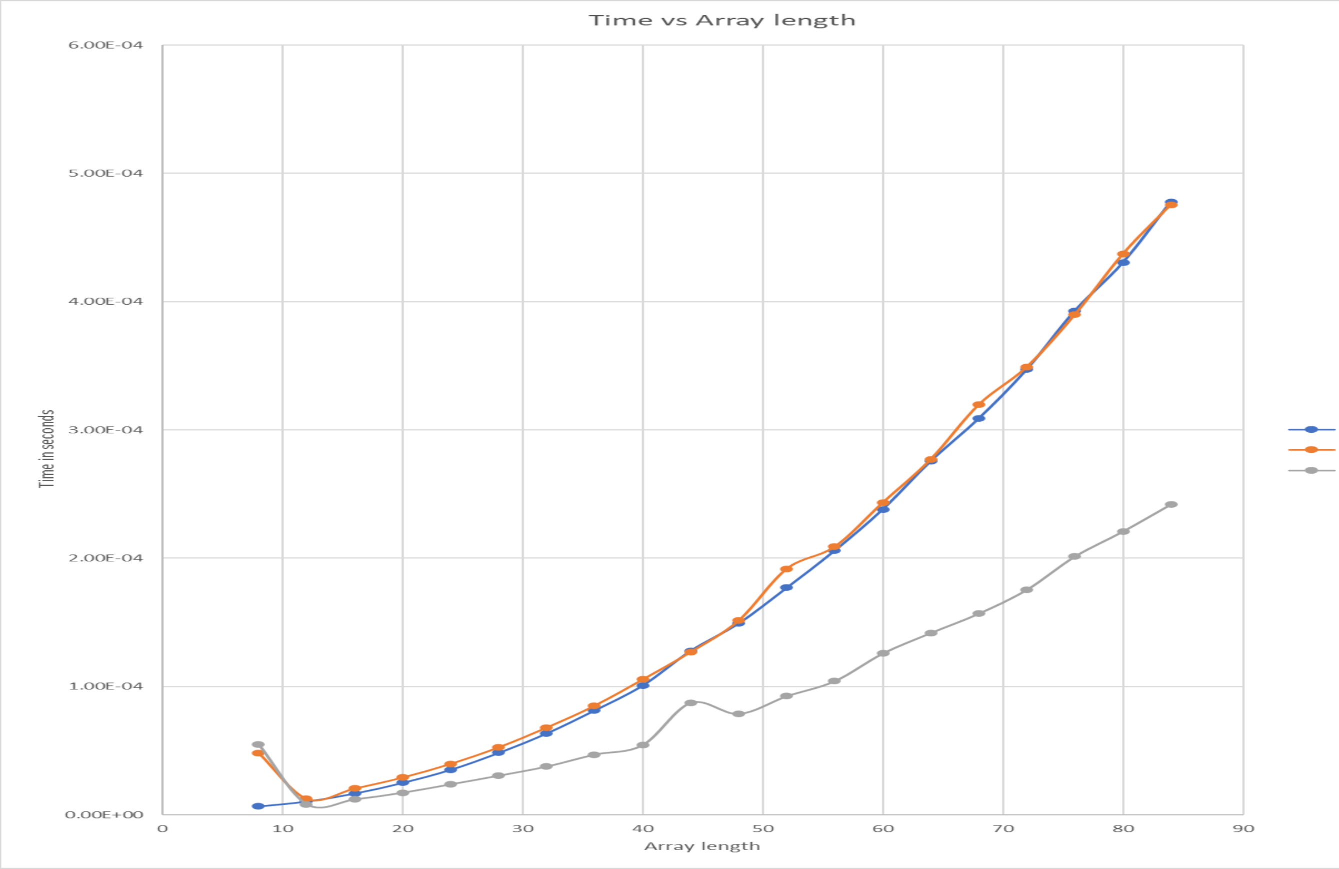
a.



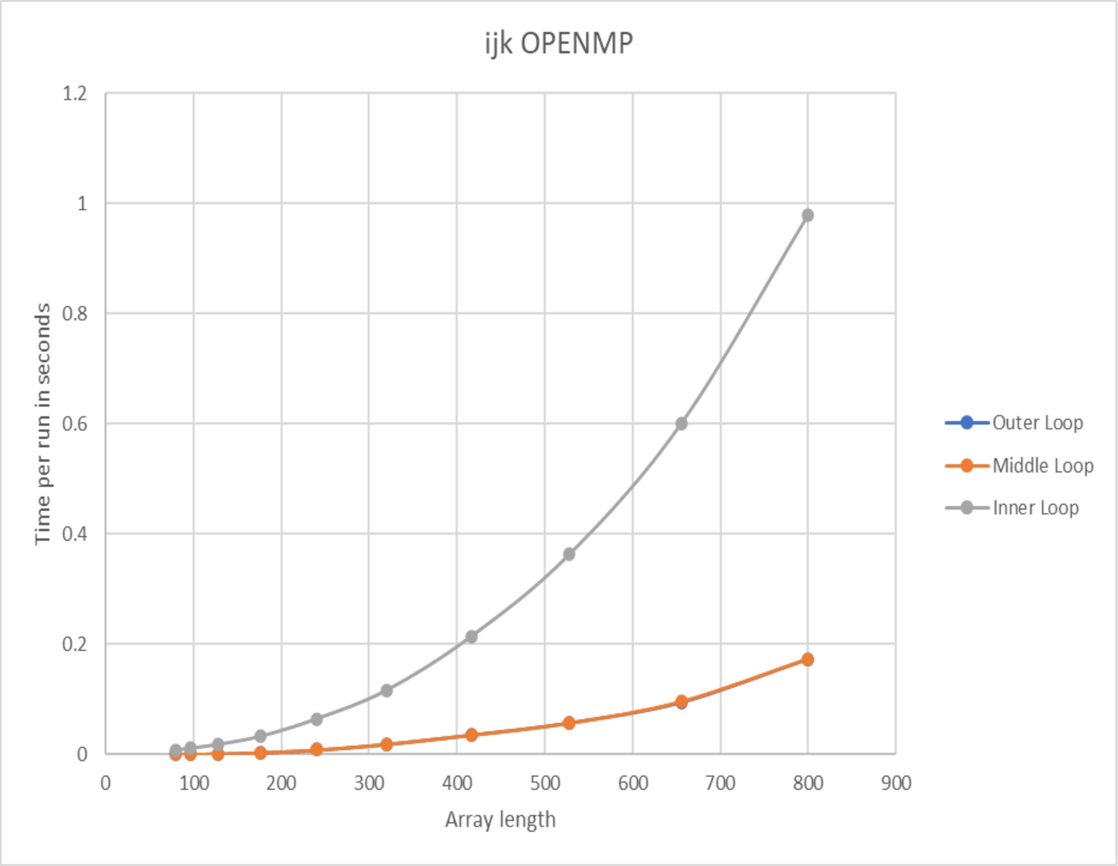
```
OpenMP race condition print test
omp's default number of threads is 4
Using 4 threads for OpenMP
Printing 'Hello world!' using 'omp parallel' and 'omp sections':

    lelH World!o

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

    wHello orld!
```

b.

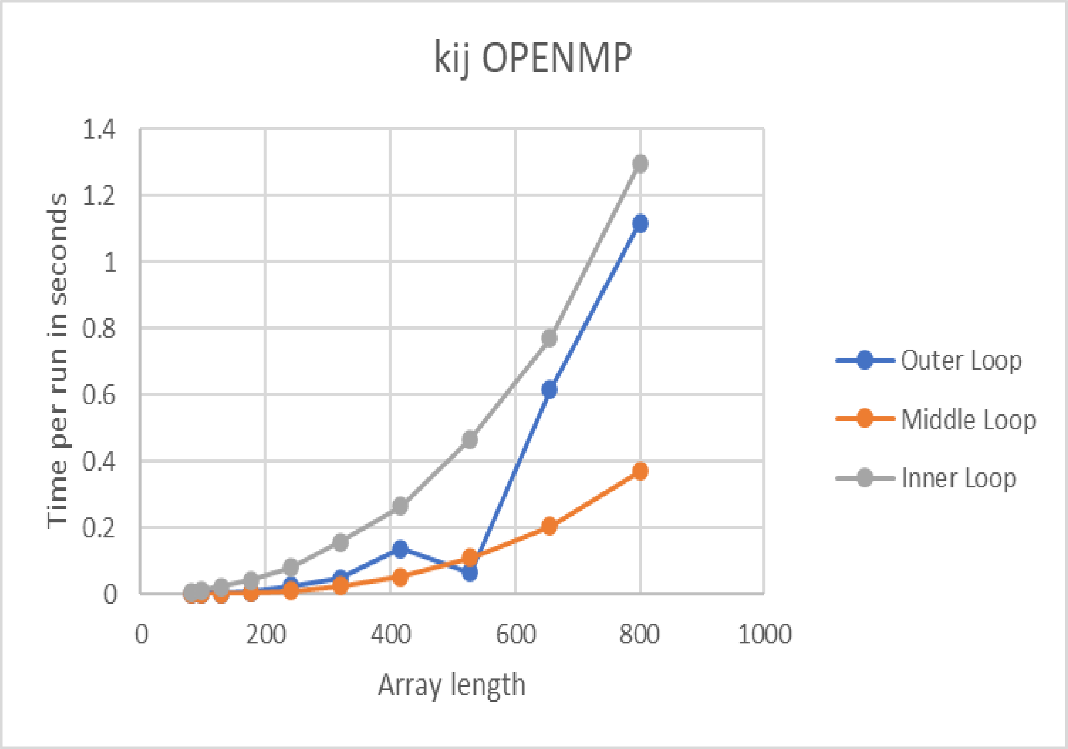
	Single Thread	Two Threads	Four Threads
Pthreads overhead (microseconds)	0.0347	36.01	59.04
Openmp overhead (microseconds)	.0946	2.535	3.435



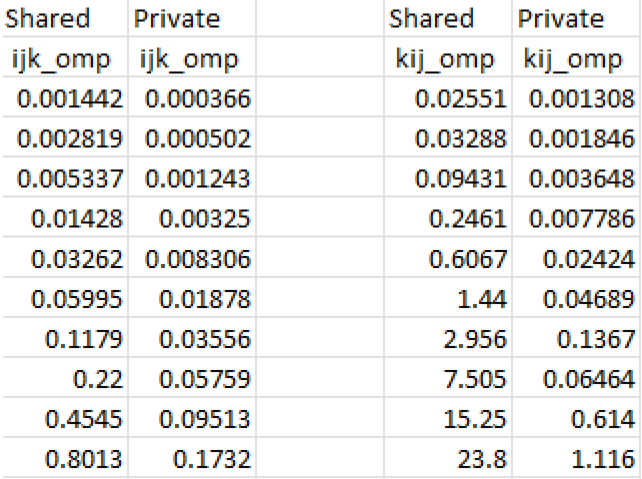
OpenMP has significantly less overhead than pthreads. This is because openMP allocates the threads at the beginning of the run. It is able to quickly pick through the list of already created threads when needed in the code. Meanwhile, pthreads is forced to create the threads while the code is running, requiring significantly more time (aka overhead) to create and task the threads while the code runs.(You can also observe the breakeven point for 4 threads vs 1 and 2 early on in the data at an array length of 10. Based on other runs the breakeven point for 2 vs 1 is also around an array length of 40)



c. MMM performance using openMP for at the outer, middle, and inner loops for ijk. The difference in performance between the outer and middle loop are negligible as nothing is actually being run in the outer loop besides the 2 inner loops. This means we get nearly the same performance between the 2. The performance when parallelizing the inner loop is significantly degraded compared to the outer or middle as we are no longer running the work contained in the middle loop (not including the inner loop) in parallel.



MMM performance using openMP for at the outer, middle, and inner loops for kij. In the case of kij the performance is best when parallelizing the middle loop. The performance when parallelizing the inner loop is significantly degraded compared to the middle, however it is only slightly worse than when parallelizing the outer loop.

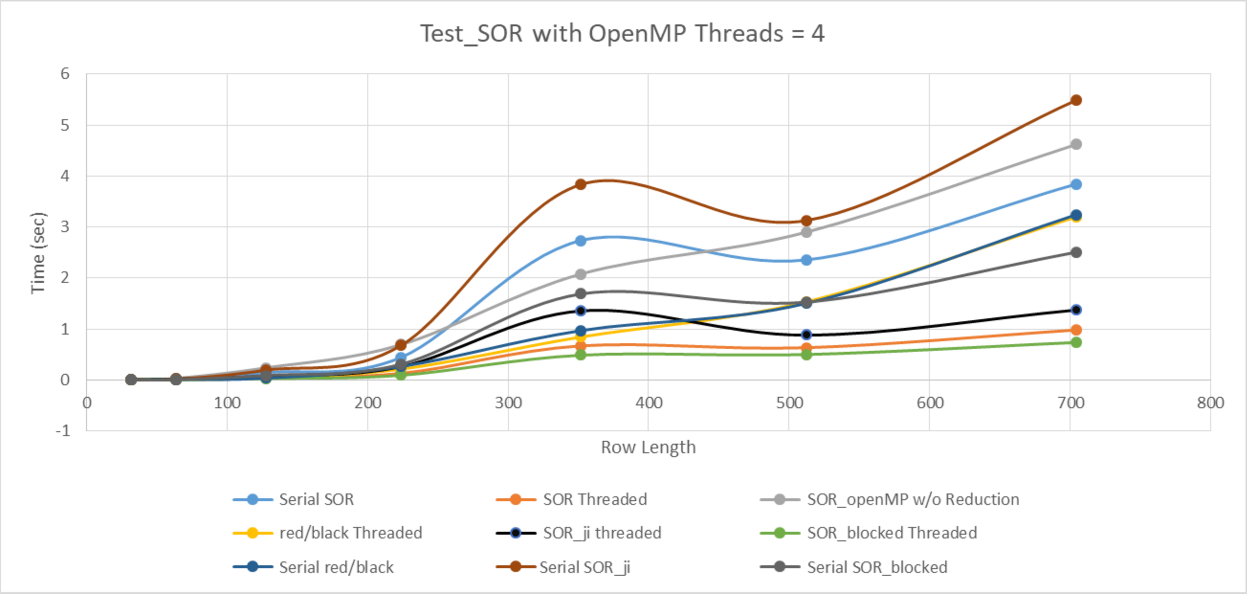


Shared	Private		Shared	Private
ijk_omp	ijk_omp		kij_omp	kij_omp
0.001442	0.000366		0.02551	0.001308
0.002819	0.000502		0.03288	0.001846
0.005337	0.001243		0.09431	0.003648
0.01428	0.00325		0.2461	0.007786
0.03262	0.008306		0.6067	0.02424
0.05995	0.01878		1.44	0.04689
0.1179	0.03556		2.956	0.1367
0.22	0.05759		7.505	0.06464
0.4545	0.09513		15.25	0.614
0.8013	0.1732		23.8	1.116

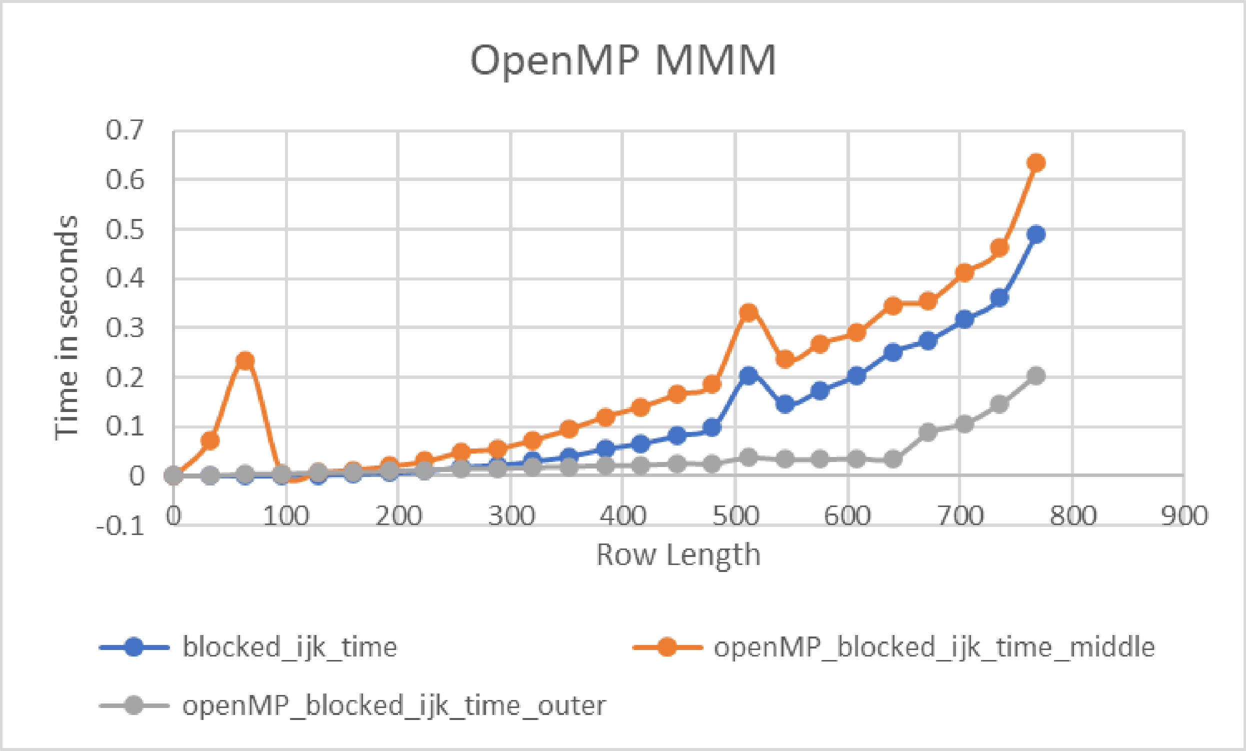
Due to the significant difference between the performance of the shared and private runs, the utility of a graph is marginal. As shown by the data presented here, sharing the i,j,k, and sum variables leads to significant to massive performance degradation of the openMP MMM. This can be attributed to the necessitated mutexing and barriers openMP uses to preserve the integrity of the variables between runs. Openmp is forced to pause and wait for other runs to complete between threads in order to preserve the integrity of the data stored in those variables. This leads to massive performance loss even when compared to the standard non parallel runs of MMM.

All data for this section of the lab can be found in Lab6Q1c.xlsx

2.



Overall the openMP threaded variants of the code are beating out the serial equivalents. In fact, the openMP variant outperforms the serial SOR by nearly 4x. The only unexpected results is from the the serial vs threaded red_black SOR. It looks like the break even point for the red/black SOR is at a row length of about 700. The threaded variant of the code should start beating out the serial version after this point in time.
Code is available in test_L62a.c



The results of applying openMP on an optimized blocked MMM code is quite interesting. When the number of threads is below 10, no real improvements can be seen; In fact, openMP performance overall is much much worse than that of the serial equivalent. However, once the number of threads is increased, a drastic improvement over the serial code can be seen as shown above. However, when the code is improperly optimized (in this case seen by the results of time_middle) it can actually lead to degradation of performance when compared to the results of the blocked serial method.

Code is available in test_L62b.c