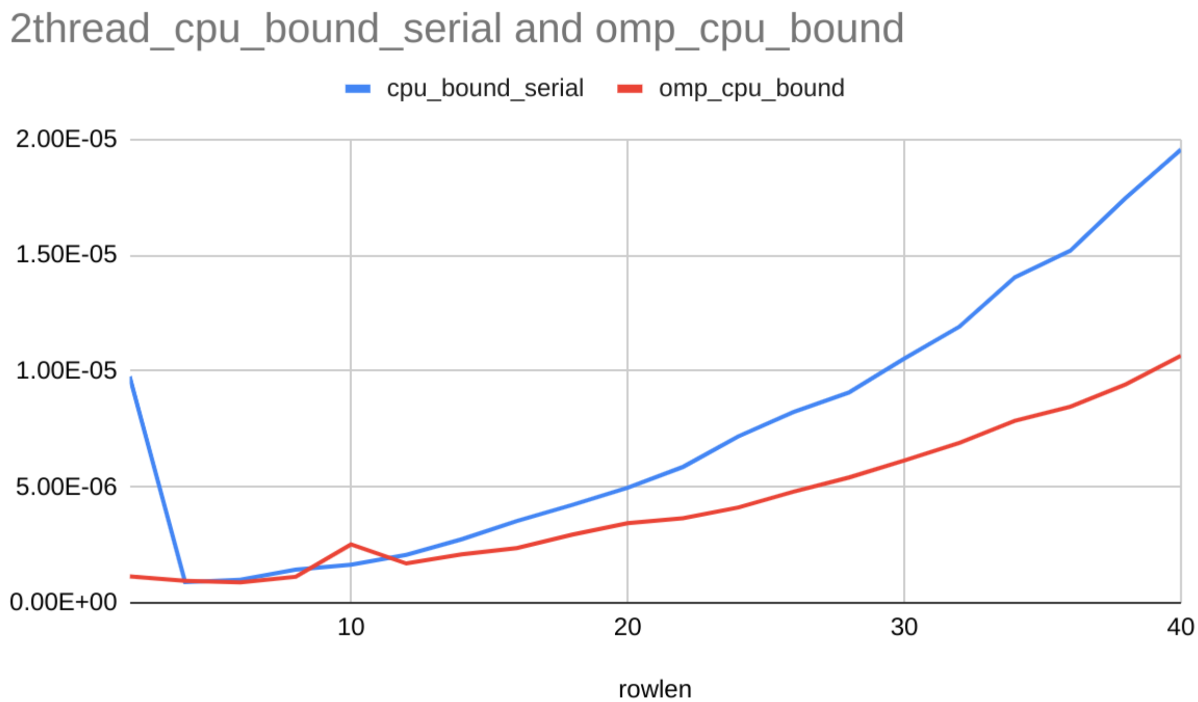
EC527
Lab6 report
Bin Xu
Seyed Reza Sajjadinasab

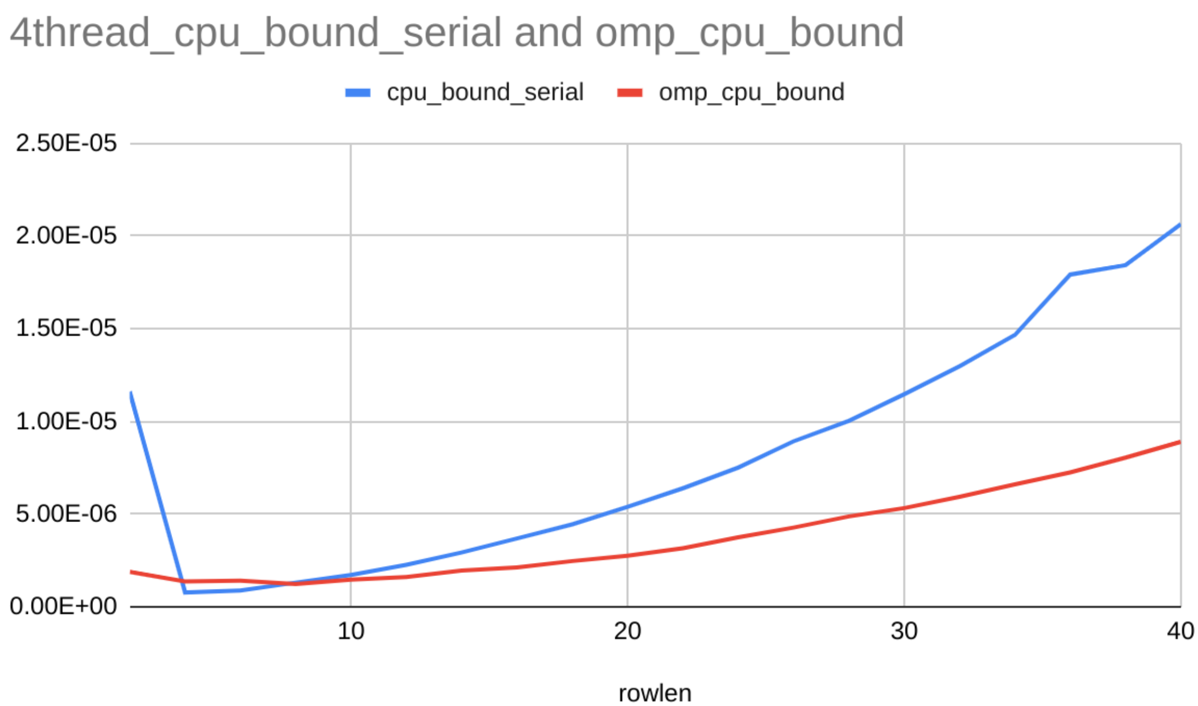
Part 1
1a.

```
char * chunk;
chunk = (char*) malloc( size: 20*sizeof (char));
strcpy( dest: chunk,  src: "Hello World!");
#pragma omp parallel for
for (int i = 0; i < omp_get_num_threads(); i++) {
    printf( format: "%c", chunk[i]);
}
printf( format: "\n\n");
```

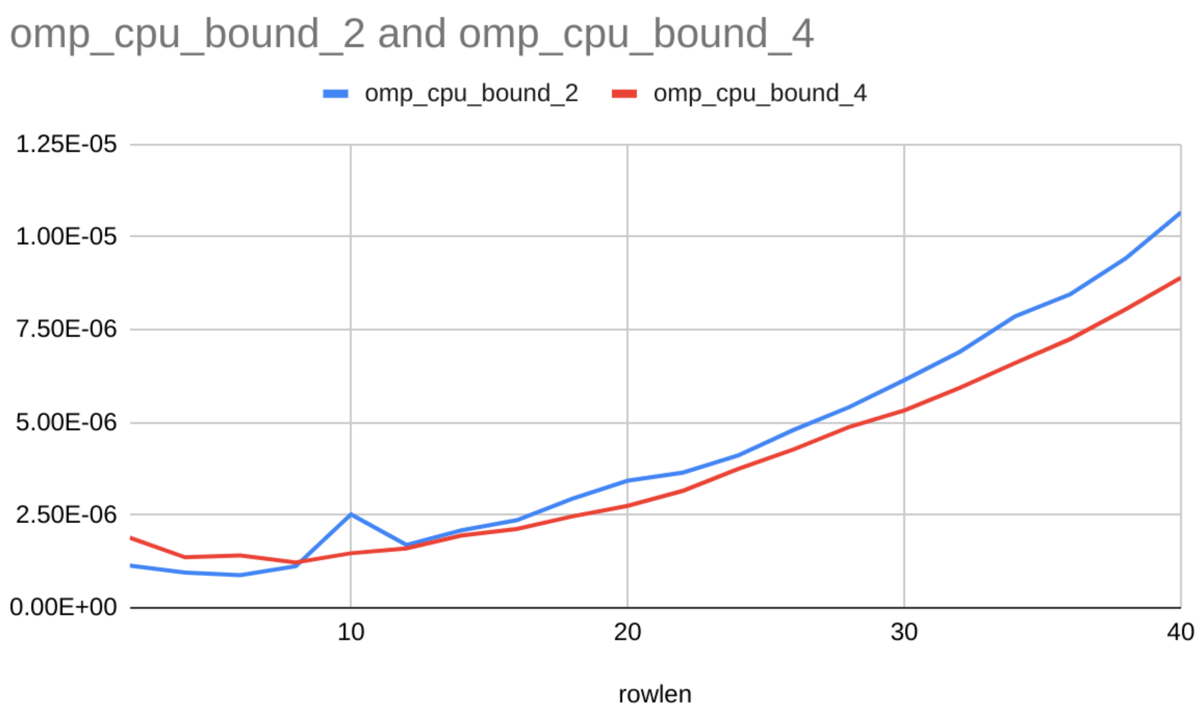
```
xu842251462@xu842251462-XPS-15-9520:~/CLionProjects/EC527_Lab6$ OMP_NUM_THREADS=20 ./test_omp
OpenMP race condition print test
omp's default number of threads is 20
Using 20 threads for OpenMP
Printing 'Hello world!' using 'omp parallel' and 'omp sections':

    eoll!r olWHd

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

    Hldeol!r Wol
```

1b.
Below are the charts for cpu bound

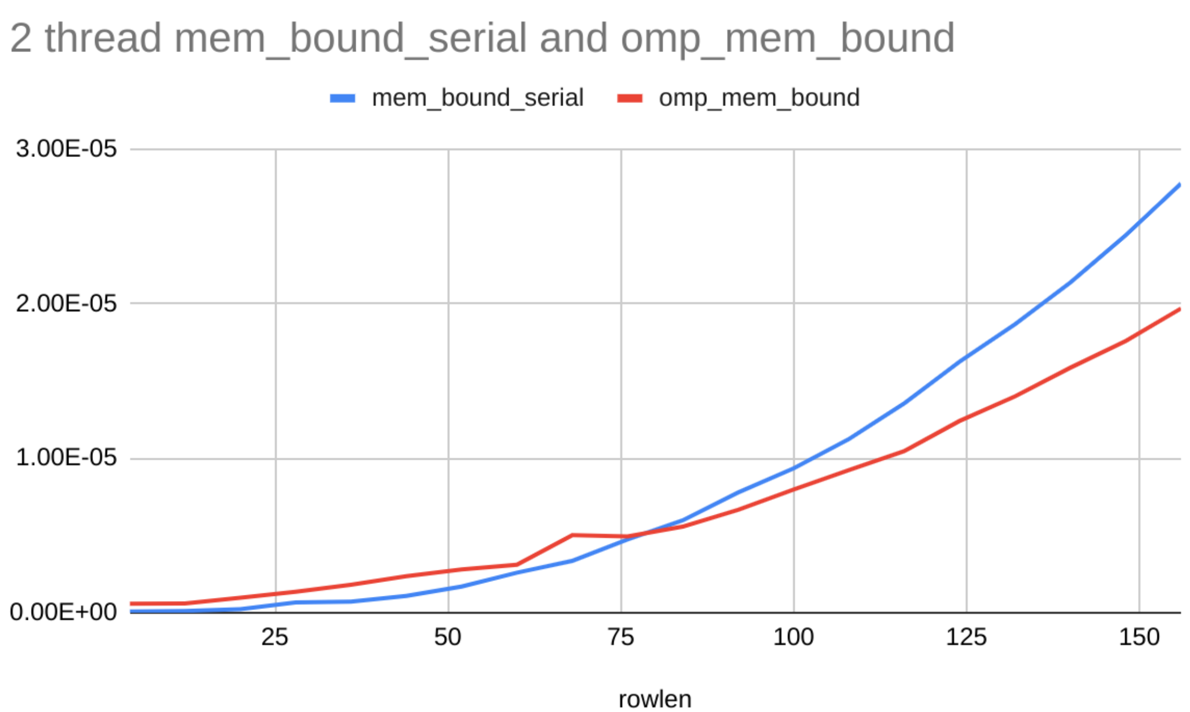




For the cpu bound, the break-even pointer in the 4-thread version is at around 8, at 1.25E-06 seconds. In the 2-thread version, the break-even point is not clear. Before the break-even point, the serial version of code is slightly faster. However, after break-even point omp version works better in bigger size operation.

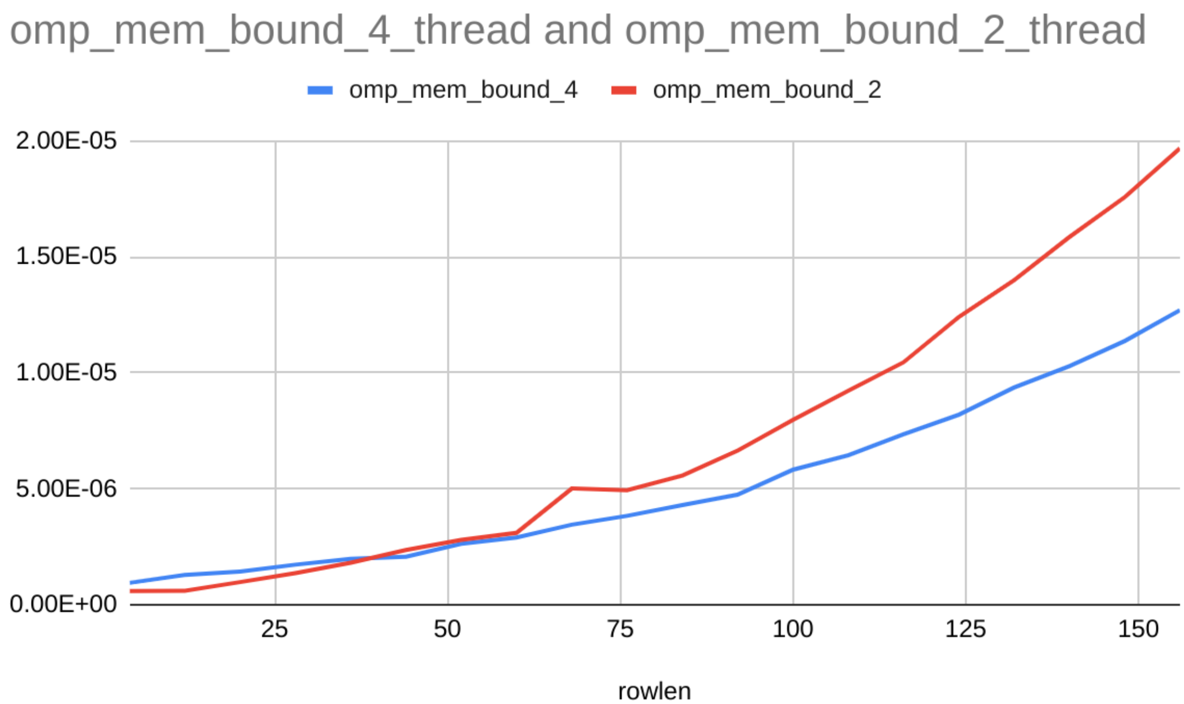


Below are charts for the mem bound



For the memory bound, the break-even pointer in the 4-thread version is at around 68, at 3.50E-06 seconds. In the 2-thread version, the break-even point occurs at around 78, at around

5.00E-06. Before the break-even point, the serial version of code is slightly faster. However, after break-even point omp version works better in bigger size operation.



From the above chart, we compare the omp_mem_thread4 to omp_mem_thread2. We find at a smaller size range from 0 to 12 the 4 thread version is around two times slower than the 2 thread version.

We applied Amdahl’s law to where two threaded was 2x faster than 4 threaded. In very small size (4) we saw this behavior, but calculating p resulted in a negative number. Which makes sense. In this area, we can assume that the difference between two versions is the overhead. That is (9.43-5.86)E-6 = 3.57 for memory overhead.

We can apply Amdahl's law after the break even point. Applying it to the break even point we have: p=0. Which means that everything is the overhead, so the value reported in this part for time is equivalent to the overhead of creating threads. The values for the time at this point had already been reported.

1c.

For the above chart, we use private() for variable scope in Omp_parallel, we also use this data for **outer loop** result.

rowlen	ijk	ijk_omp	kij	kij_omp
20	3.20E-06	0.0001924	4.59E-06	0.0002276
120	0.0006228	0.0003434	0.0009919	0.0006685

260	0.006918	0.00381	0.008648	0.003371
440	0.0406	0.02195	0.03263	0.014
660	0.1509	0.0661	0.09285	0.04938
920	0.4078	0.1961	0.2574	0.145
1220	0.9526	0.4223	0.7091	0.3508
1560	2.229	1.074	1.497	0.6343
1940	8.625	2.655	2.842	0.9687
2360	30.32	9.137	5.531	2.189

For the above data, we use shared() for the variable scope in Omp_parallel.

shared				
rowlen	ijk	ijk_omp	kij	kij_omp
20	2.93E-06	0.0001624	3.49E-06	0.0001451
120	0.0005357	0.0003329	0.0007974	0.000288
260	0.006224	0.00364	0.007418	0.002357
440	0.03348	0.01179	0.03161	0.01063
660	0.1179	0.03438	0.1021	0.02457
920	0.3497	0.09671	0.2657	0.05971
1220	0.7878	0.2208	0.5989	0.1395
1560	1.771	0.5073	1.269	0.2932
1940	6.663	1.618	2.641	0.5822
2360	27.05	6.18	5.095	1.169

In OpenMP, shared() and private() will have influence on the performance of matrix multiplication. From the above data, when moving all the variables from private() to shared(), the performance will be improved. Because all variables run parallelly in shared memory.

placing ‘parallel’ in **middle loop**

rowlen	ijk	ijk_omp	kij	kij_omp
20	3.40E-06	0.0002175	1.05E-05	0.0002133
120	0.0006351	0.0004145	0.002129	0.0004803
260	0.008188	0.003694	0.01867	0.00472
440	0.04466	0.01725	0.04679	0.0135
660	0.1403	0.06282	0.15	0.0593
920	0.412	0.1877	0.4102	0.1322

1220	0.9872	0.5021	0.9491	0.3146
1560	2.143	1.328	2.138	0.6721
1940	6.652	2.222	4.466	1.371
2360	32.1	12.58	8.563	2.71

placing ‘parallel’ in **inner loop**

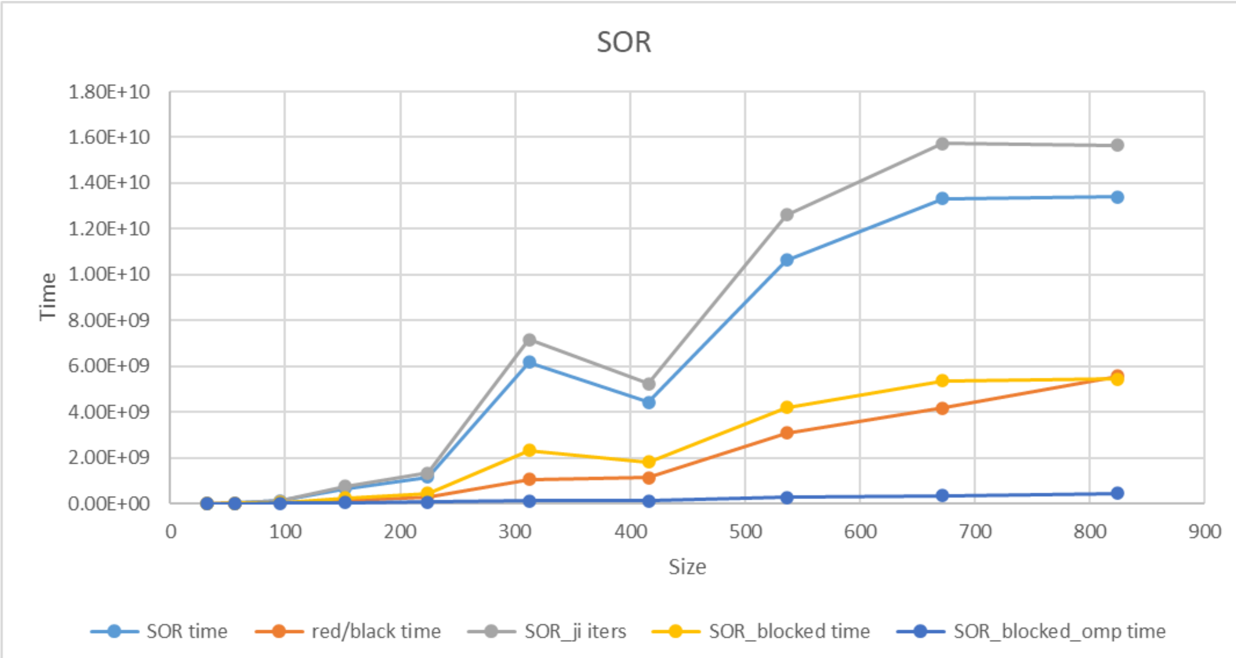
rowlen	ijk	ijk_omp	kij	kij_omp
20	3.60E-06	0.0007684	6.72E-06	0.0006919
120	0.0005918	0.02416	0.001183	0.02031
260	0.007788	0.1043	0.01279	0.08822
440	0.04556	0.3178	0.04033	0.3562
660	0.1447	0.7728	0.1218	0.6549
920	0.401	1.98	0.3157	1.367
1220	0.8983	4.443	0.731	2.213
1560	2.05	7.824	1.424	4.217
1940	10.3	13.03	3.064	6.327
2360	32.46	19.88	6.296	9.611

From above the data tested from outer loop to middle loop, then to inner loop. The performance is dropped accordingly. Because less variables are shared between different threads. In parallel computing, the private variables have to be created in different threads, the performance will be dropped.

Part2

2a.

We test the SOR_block version. From the above data, OpenMP version of SOR_block is faster than the original SOR_block. For the bigger matrix size, OpenMP has more advantage on the parallel programming.



As we can see, the OpenMP version is much better than others.

2b.

The MMM version shows different behaviors for different threads. In all numbers of threads, for smaller sizes the blocking version was better. However, for larger sizes the non blocking was better.

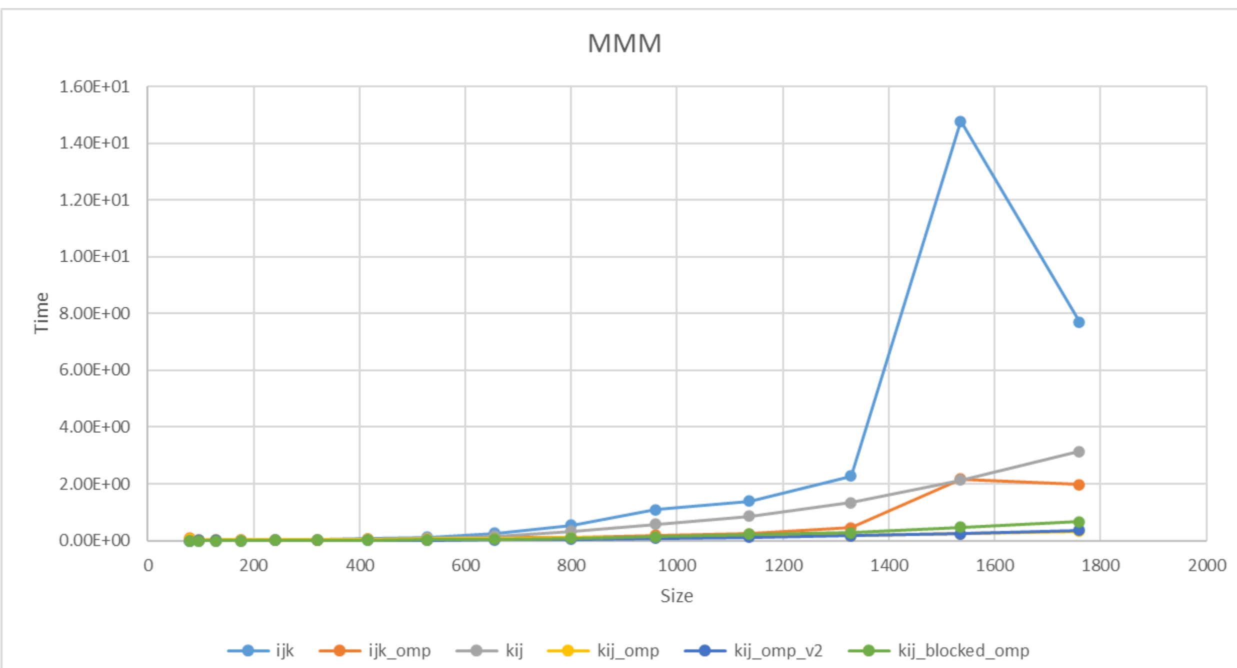
To make the comparison even more distinctive we use another non-blocking version that parallelizes the two outermost loops. In this version for Num_threads=10, it was the best, but for smaller e.g. Num_threads=4 or bigger e.g. Num_threads=20.

Results are shown bellow:

OMP_NUM_THREADS=20

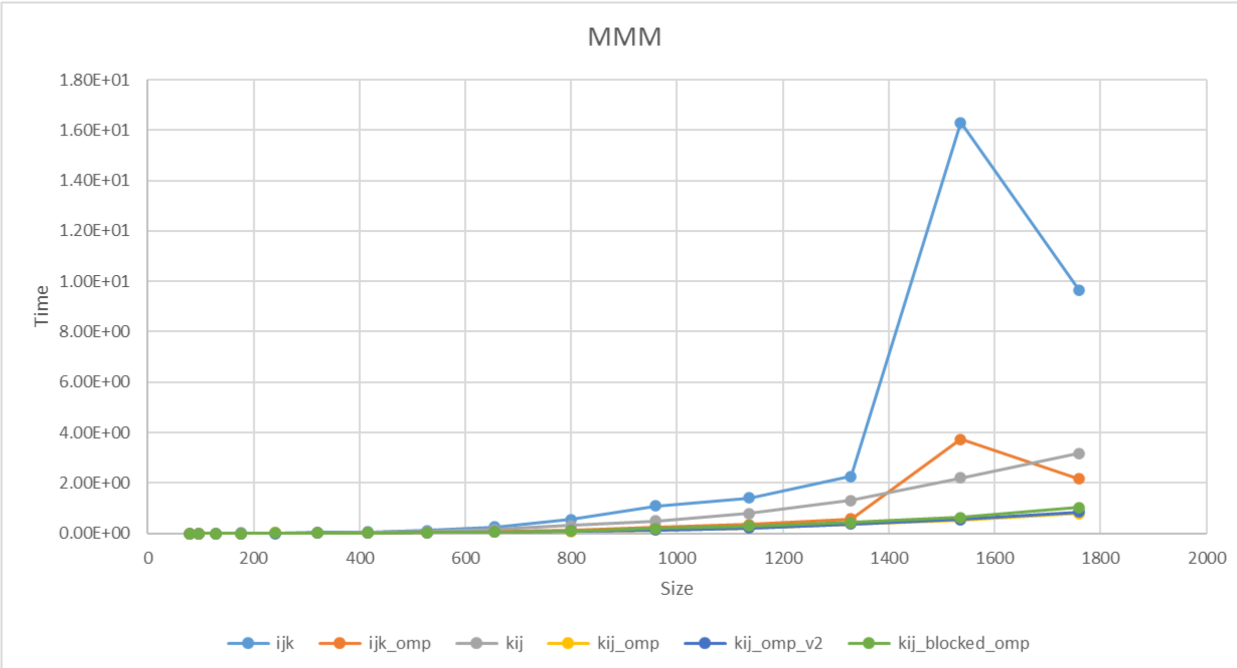
rowlen	ijk	ijk_omp	kij	kij_omp	kij_omp_v2	kij_blocked_omp
80	4.14E-04	0.08723	5.27E-04	0.07086	0.002724	0.000438
96	7.13E-04	0.01927	1.04E-03	0.02339	0.004176	0.000938
128	2.50E-03	0.03741	2.38E-03	0.03103	0.006342	0.001439
176	5.02E-03	0.03691	7.87E-03	0.03394	0.001653	0.002766

240	1.02E-02	0.01749	1.41E-02	0.02288	0.005451	0.004224
320	3.29E-02	0.02729	1.98E-02	0.03266	0.007432	0.007436
416	5.98E-02	0.04798	3.99E-02	0.05017	0.01008	0.01491
528	1.17E-01	0.03706	8.10E-02	0.05727	0.01362	0.02639
656	2.60E-01	0.1117	1.55E-01	0.07276	0.02204	0.05187
800	5.46E-01	0.1199	3.35E-01	0.1039	0.03955	0.0827
960	1.093	0.1901	0.5844	0.1173	0.07601	0.1292
1136	1.388	0.2367	0.8525	0.1651	0.1149	0.2174
1328	2.272	0.4504	1.34	0.1747	0.1772	0.2729
1536	14.77	2.164	2.127	0.2483	0.2457	0.4737
1760	7.704	1.967	3.134	0.3371	0.3633	0.6607



OMP_NUM_THREADS=4

rowlen	ijk	ijk_omp	kij	kij_omp	kij_omp_v2	kij_blocked_omp
80	2.73E-04	0.000449	5.56E-04	0.001037	0.001018	0.000479
96	4.84E-04	0.000291	7.58E-04	0.000309	0.000393	0.000625
128	1.65E-03	0.001259	1.38E-03	0.000763	0.000802	0.000979
176	3.58E-03	0.001968	3.29E-03	0.001827	0.001589	0.002533
240	1.13E-02	0.005958	1.49E-02	0.003973	0.003274	0.005333
320	3.20E-02	0.01802	2.01E-02	0.01088	0.006613	0.008769
416	6.11E-02	0.01711	4.17E-02	0.02062	0.01118	0.01633
528	1.21E-01	0.03508	8.30E-02	0.0274	0.0237	0.03195
656	2.59E-01	0.06914	1.58E-01	0.04729	0.04376	0.0526
800	5.54E-01	0.1309	3.04E-01	0.08291	0.08711	0.1008
960	1.079	0.256	0.4964	0.1307	0.1393	0.1991
1136	1.401	0.3476	0.7971	0.2136	0.2143	0.285
1328	2.252	0.5708	1.302	0.3623	0.3699	0.4316
1536	16.3	3.734	2.197	0.5214	0.5428	0.6359
1760	9.635	2.17	3.181	0.791	0.8411	1.024



OMP_NUM_THREADS=10

rowlen	ijk	ijk_omp	kij	kij_omp	kij_omp_v2	kij_blocked_omp
80	2.63E-04	0.04127	3.42E-04	0.000332	0.001407	0.000548
96	4.67E-04	0.02424	6.16E-04	0.000171	0.000238	0.000715
128	1.63E-03	0.02755	1.35E-03	0.000404	0.00038	0.001401
176	3.44E-03	0.02726	3.37E-03	0.000869	0.000966	0.002731
240	1.01E-02	0.03578	9.14E-03	0.002185	0.003601	0.005625
320	3.13E-02	0.02616	2.02E-02	0.004015	0.005009	0.009208
416	6.07E-02	0.02389	4.02E-02	0.0106	0.01042	0.02048
528	1.28E-01	0.04527	8.01E-02	0.01782	0.01788	0.03513

656	2.59E-01	0.06498	1.54E-01	0.03282	0.03508	0.04932
800	5.42E-01	0.1102	2.81E-01	0.05547	0.05667	0.07368
960	1.04	0.164	0.4924	0.09419	0.09124	0.1432
1136	1.363	0.235	0.8003	0.1317	0.145	0.1835
1328	2.193	0.3721	1.289	0.2069	0.2234	0.2762
1536	14.73	1.656	1.996	0.3114	0.3093	0.4114
1760	7.666	1.032	3.162	0.4396	0.4402	0.5761