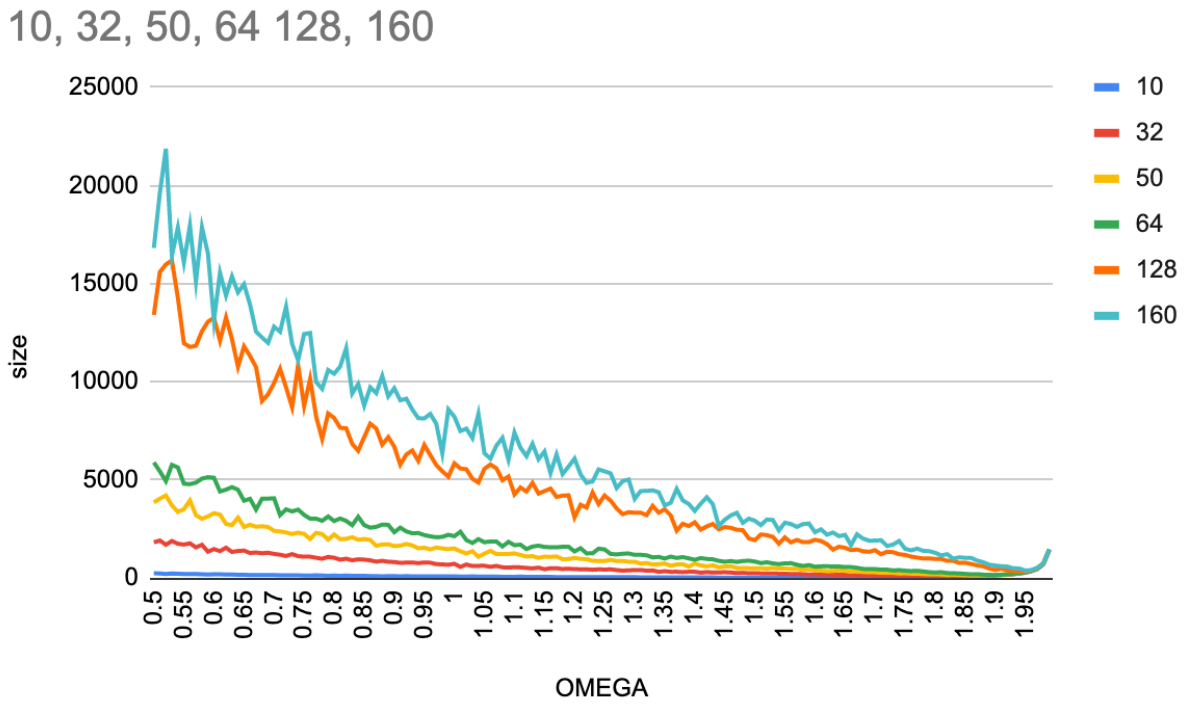
EC527
Lab5 report
Bin Xu
Seyed Reza Sajjadinasab

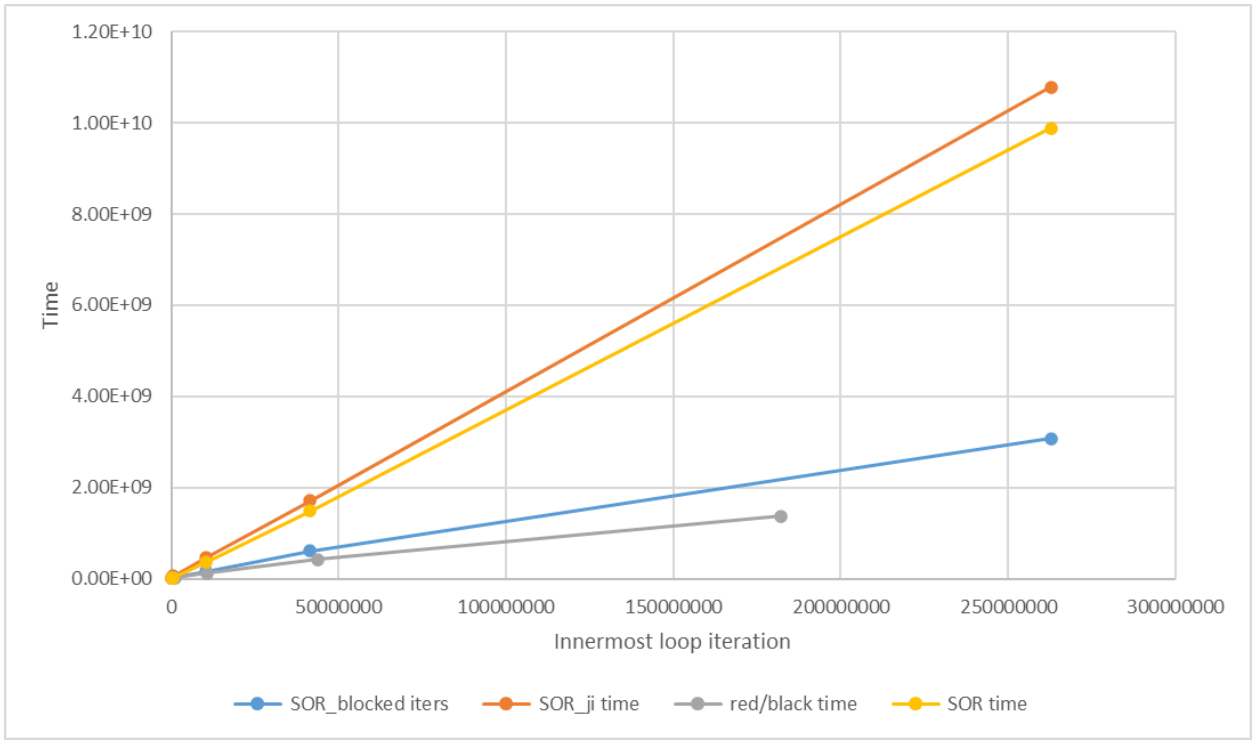
Task1
L2 cache size: 12MB



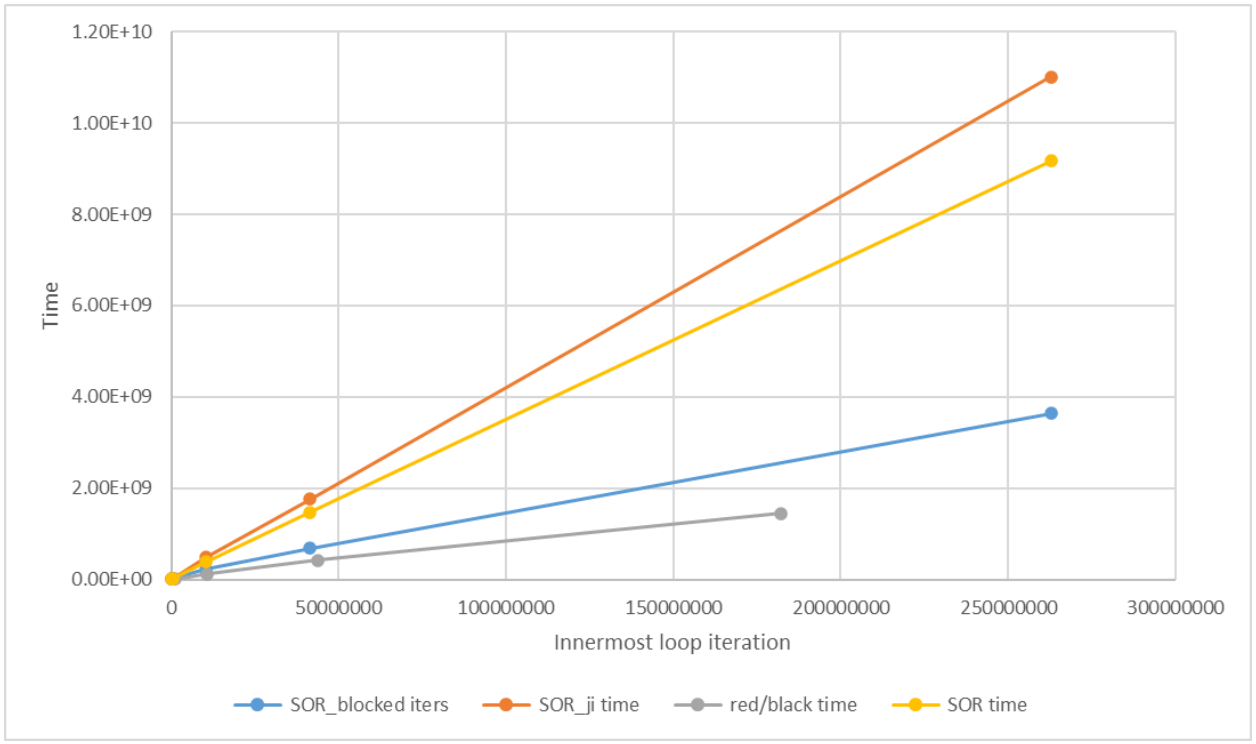
- For the chart, We test the different array sizes. The OMEGA ranges from 0.5 to 2.0. From the shape of the chart, the number of iterations decrease with the increasing OMEGA value. The coverage occurs when OMEGA reaches the optimal value.
- From the above chart, the OMEGA value increases with array size.
- OMEGA value is sensitive from 0.5 to 1.25.
- The optimal OMEGA value is close to 1.95. The number of iterations to coverage is affected by cache size. The miss rate can increase when the array size does not fit in the cache.

Task 2
As inferred from task1 we set the OMEGA to be 1.5 and 1.8, and 1.9. The below results are measured on the local machine.

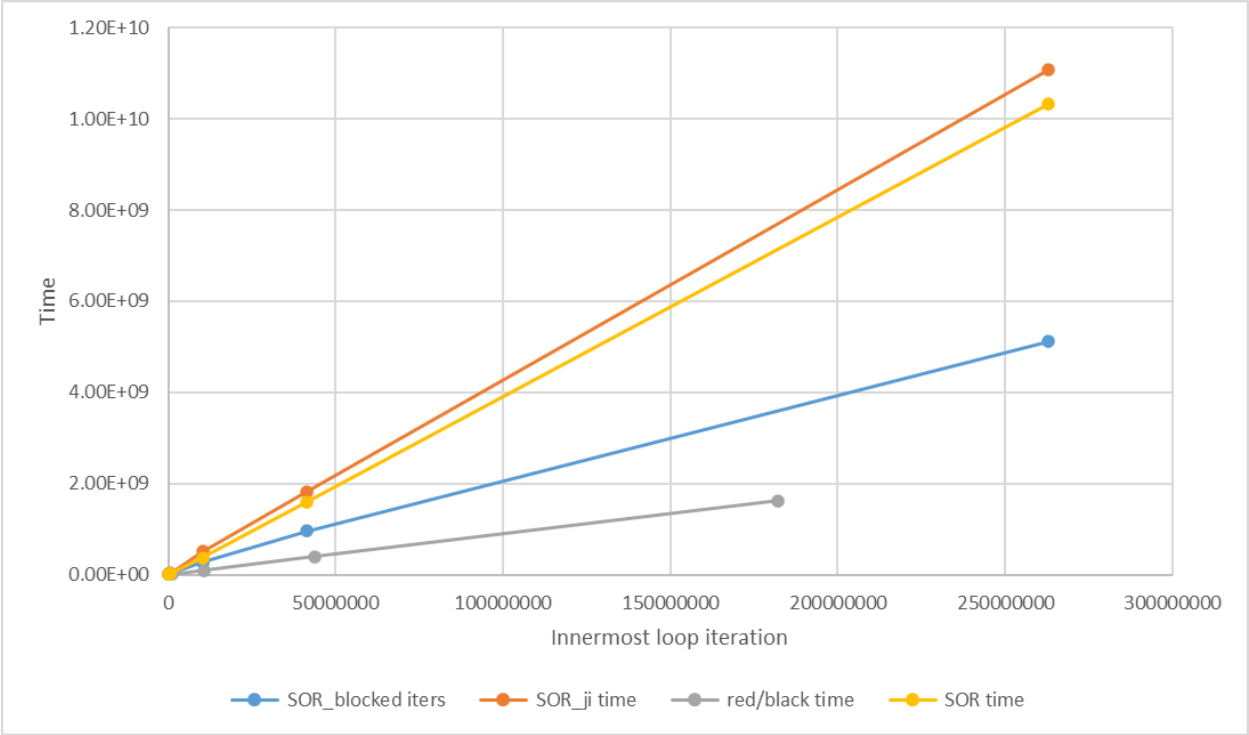
Block=4, OMEGA=1.8



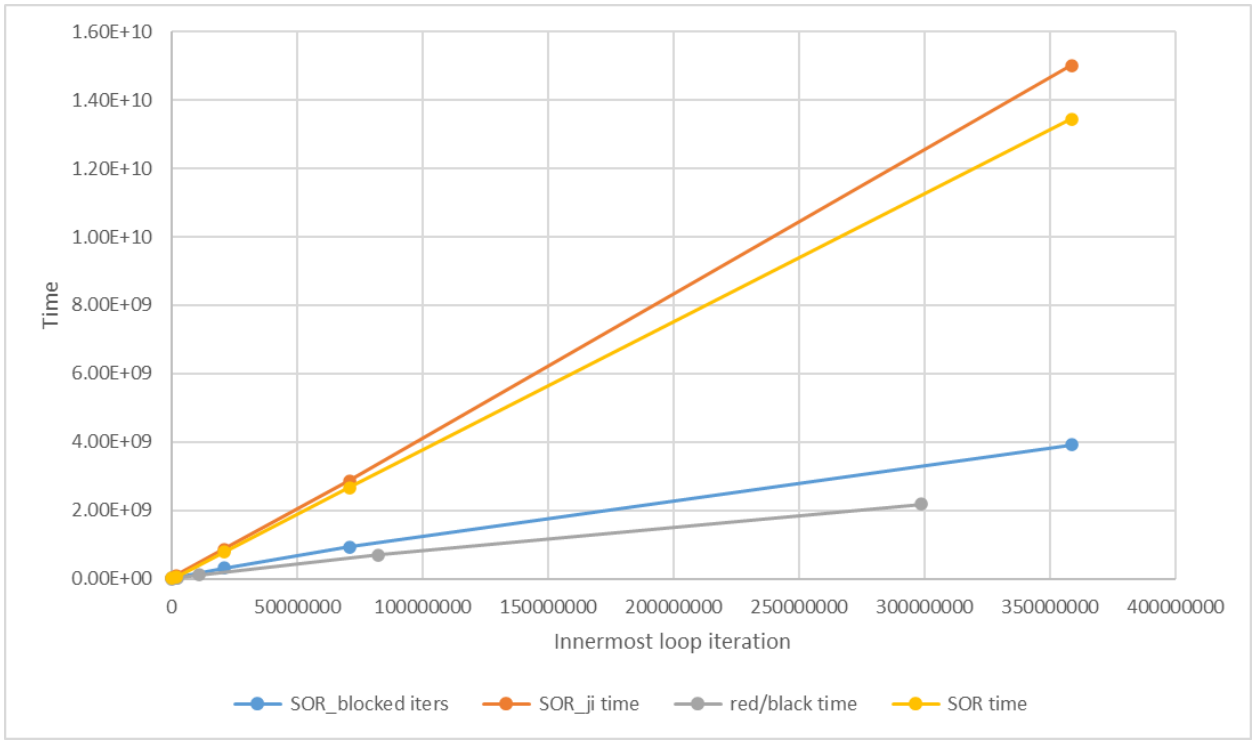
Block=8, OMEGA=1.8



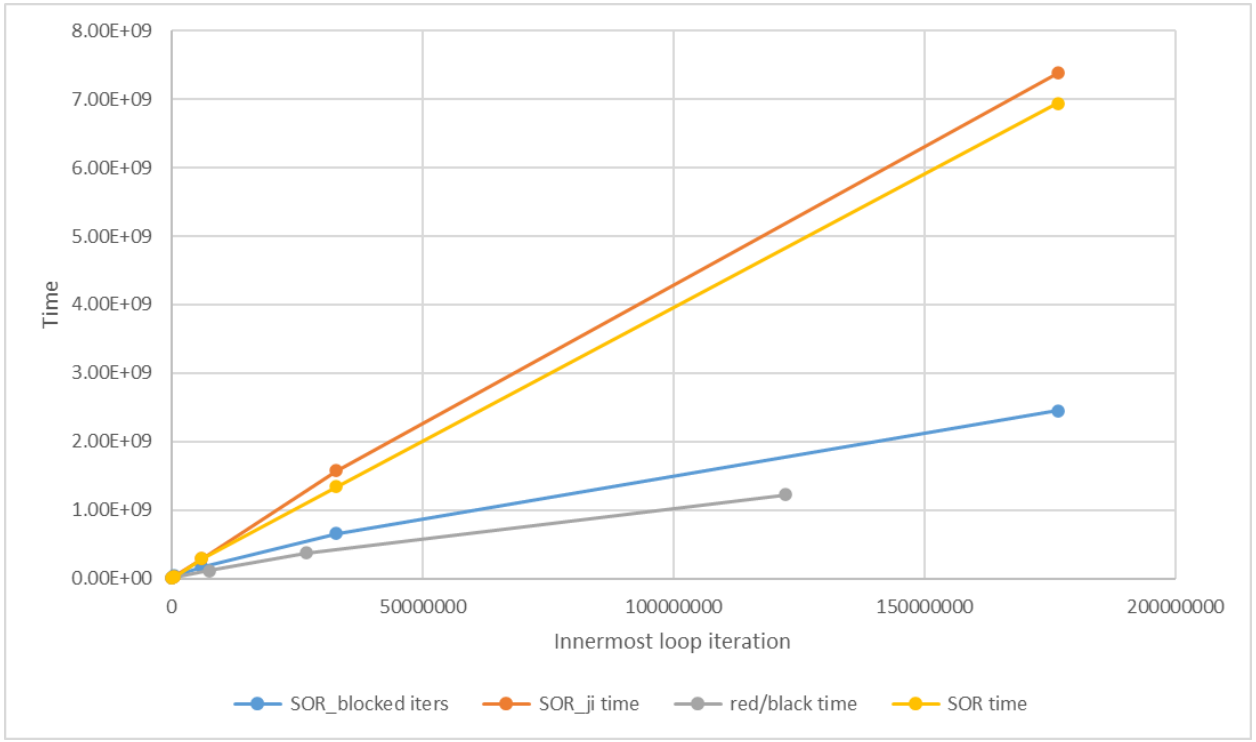
Block=16, OMEGA=1.8



Since a larger block leads to a worse performance for OMEGA 1.5 and 2 we used a block size of 4.
Block=4, OMEGA=1.5



Block=4, OMEGA=1.9

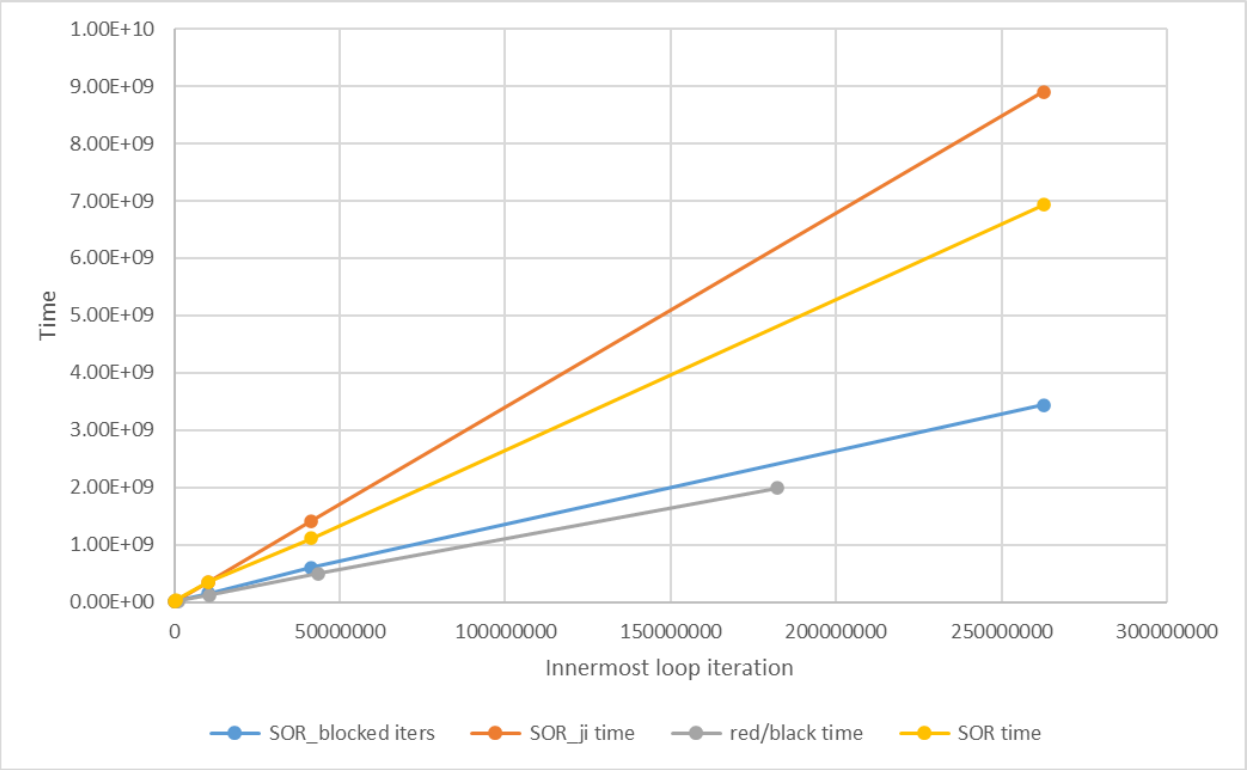


As we can see the best performance is for OMEGA=1.9 and block size of 4.

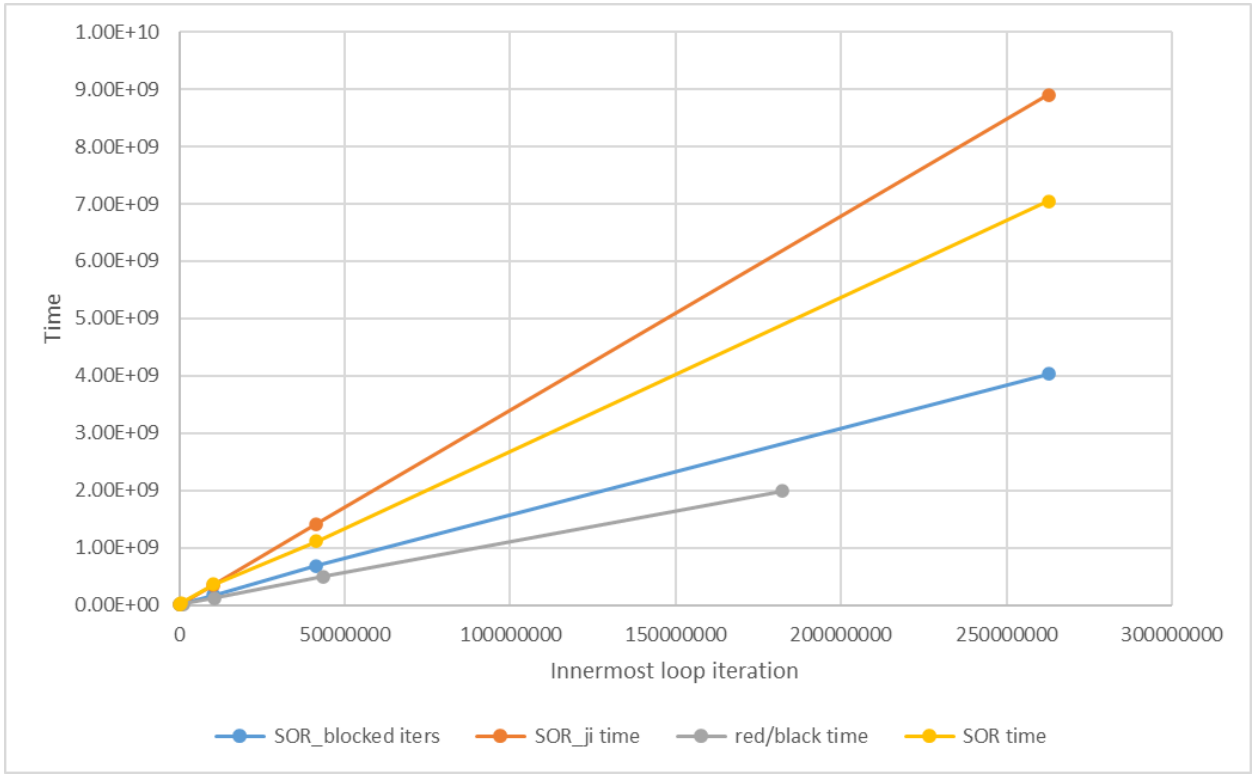
The thing about this part is that blocking is not better than the redblack method. However, They are fairly similar and way much better than the two other methods. This is because of the intrinsic superiority that redblack has. The redblack version has a better co-to-co ratio which compensates for not being able to fit into the cache as the blocking version can.

On the lab machine we have these graphs:

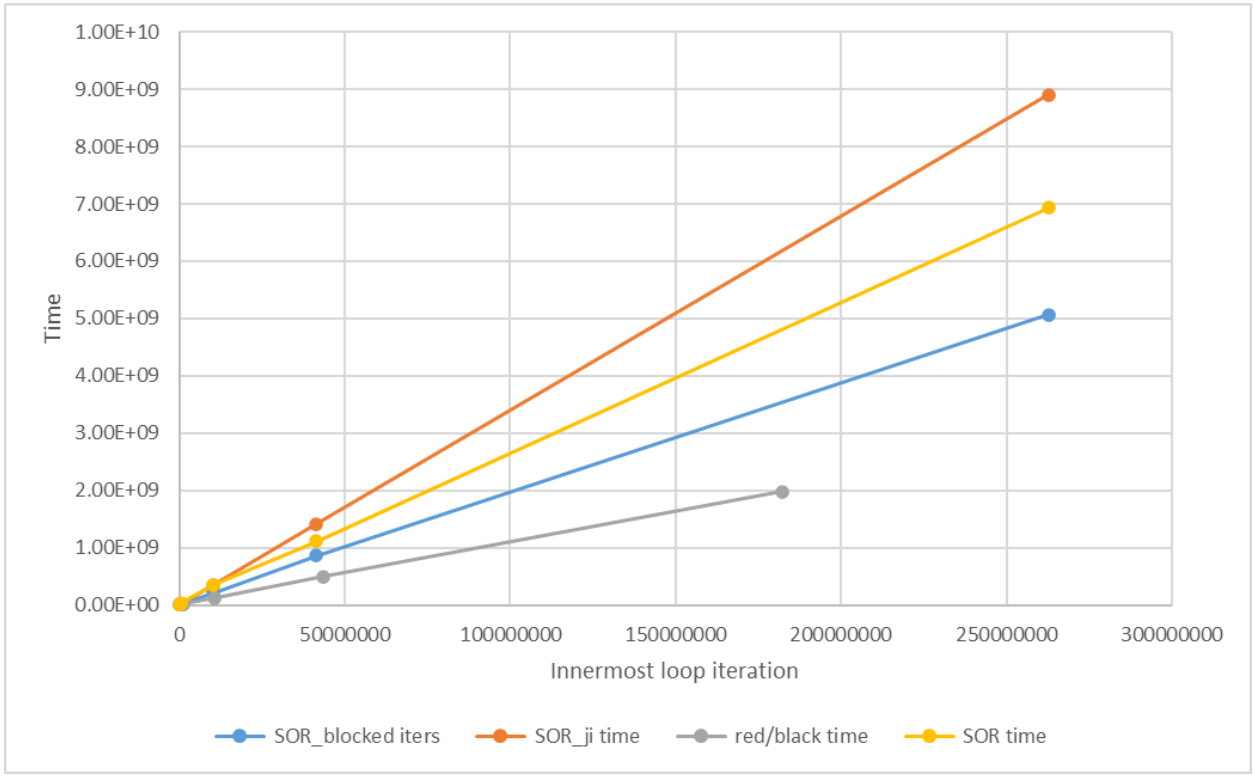
Block=4, OMEGA=1.8



Block=8, OMEGA=1.8

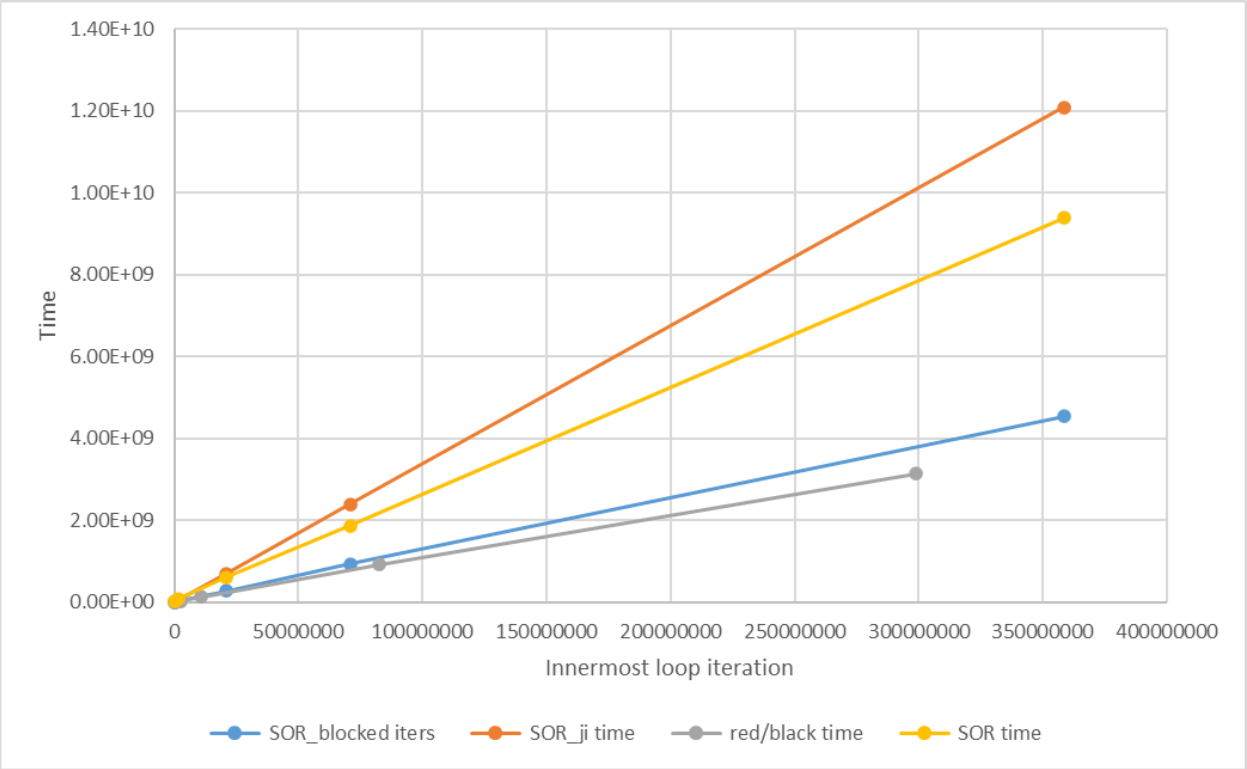


Block=16, OMEGA=1.8

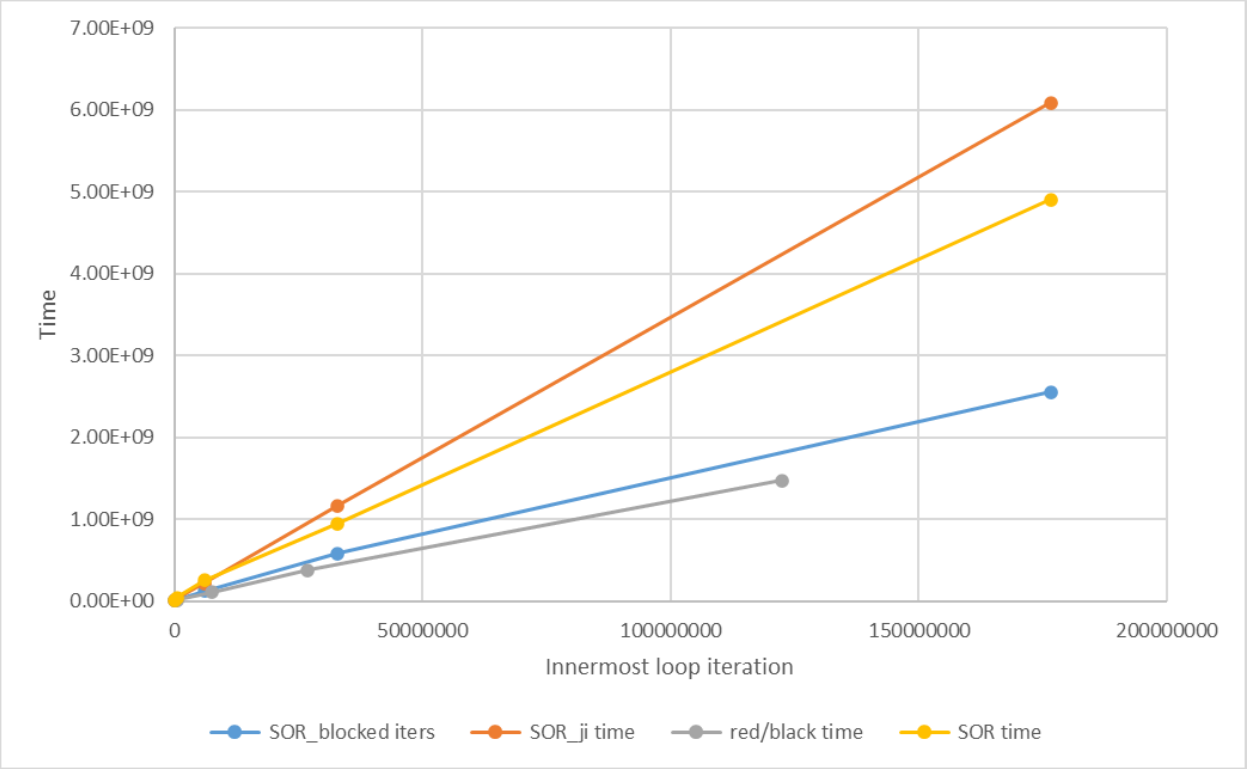


Same as the local machine the smaller block size was a better choice.

Block=4, OMEGA=1.5



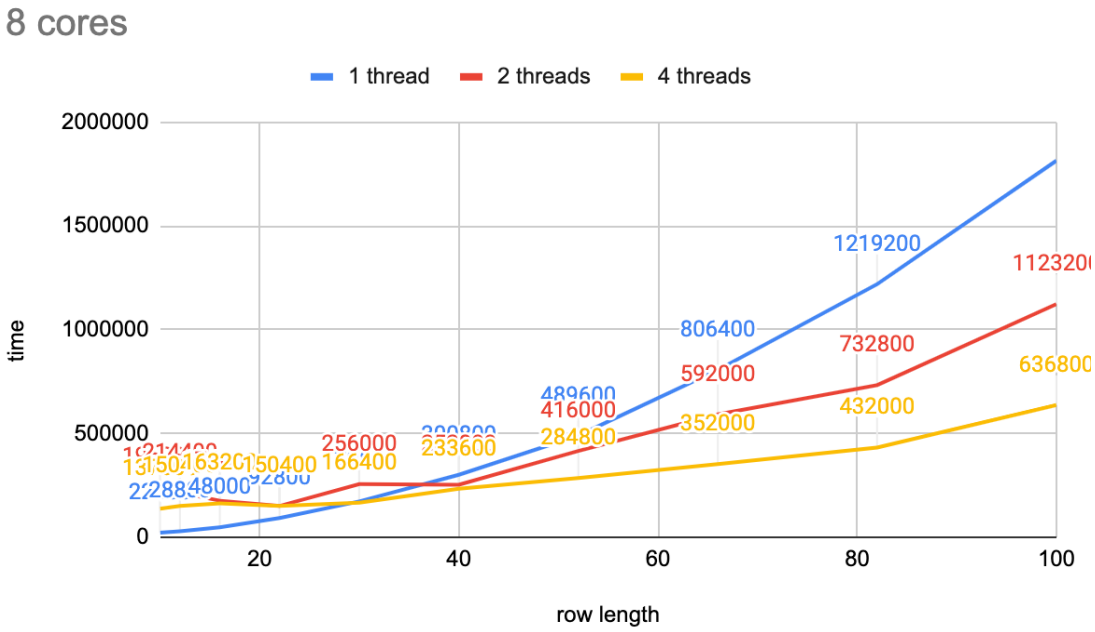
Block=4, OMEGA=1.9



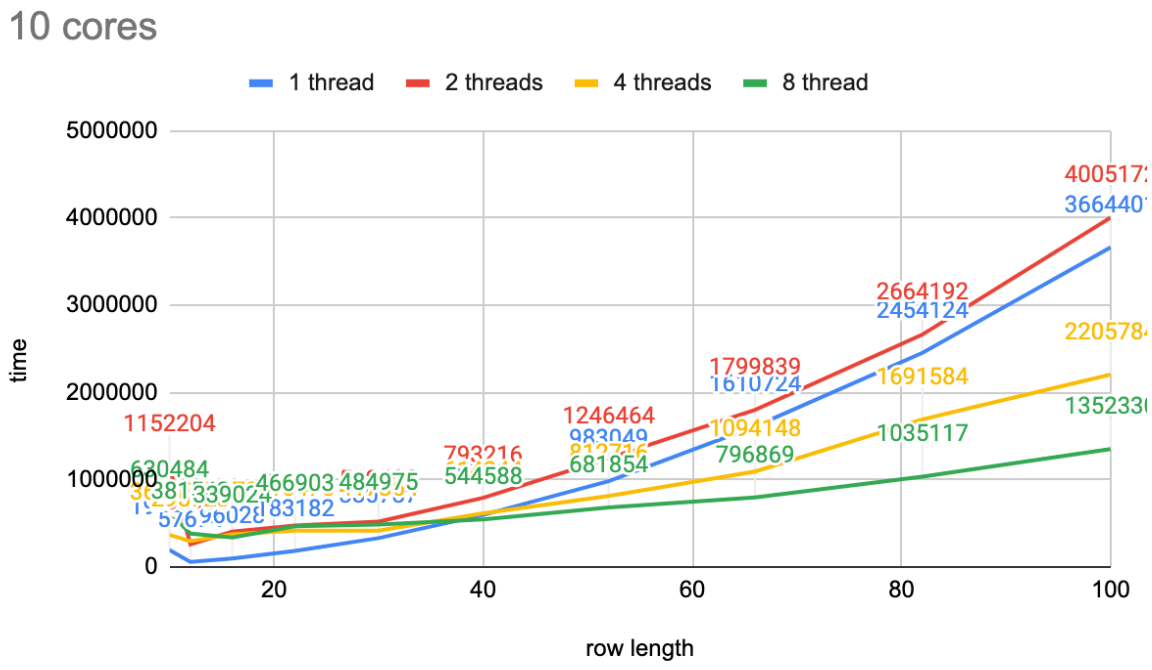
As we can see the best performance is for OMEGA=1.9 and block size of 4.

In this task we saw that increasing the block size makes everything worse. It's because of the fact that with a larger block size we have fewer blocks and as we saw in the slide the blocking version doesn't have a good co-to-co ratio in the lower numbers of blocks.

Task 3



Using the 8 cores, the baseline method takes more time than the pthread with thread number 2 and pthread number 4.

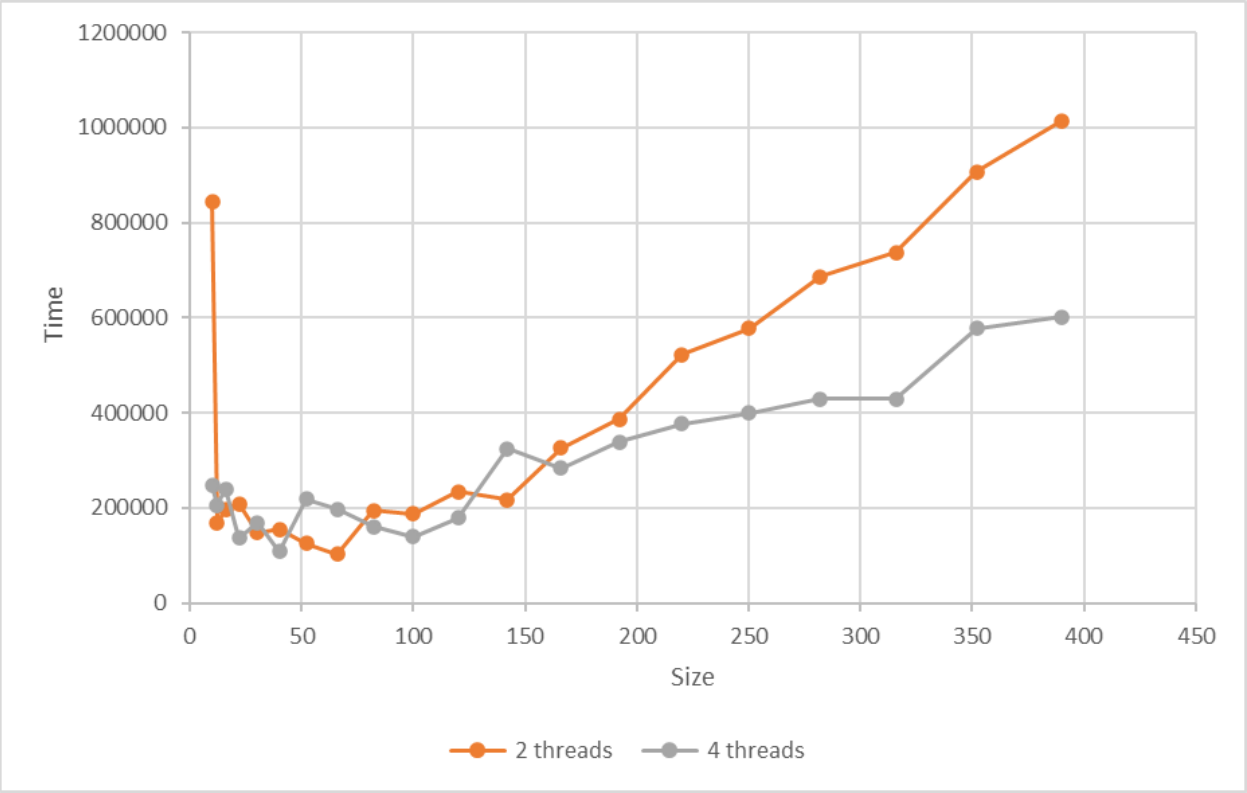


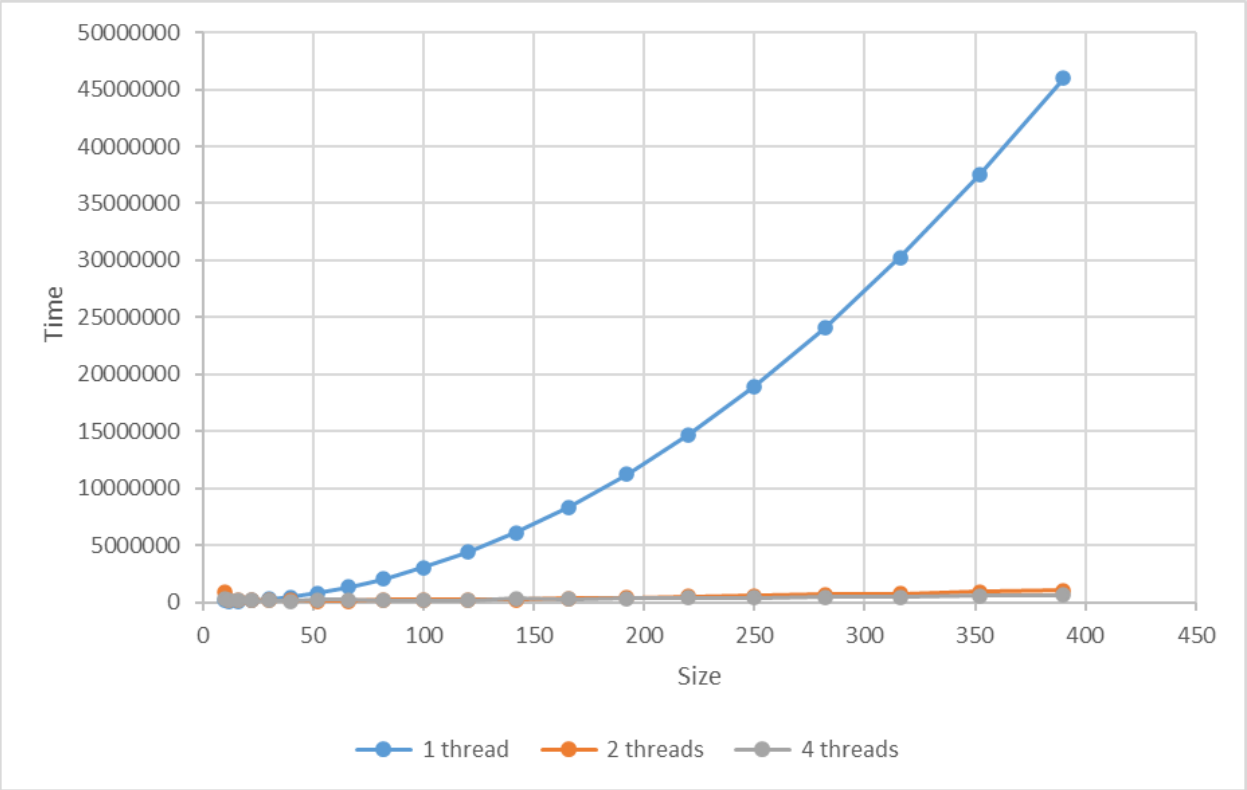
For row length of 100, the overall speedup of 4 threads to 1 threads is about 1.63. From Amdahl's law we now that this overall speedup must be $\frac{1}{(1-p)+p/s}$. Here s is 4 because we are using 4 threads.

Using this we can obtain that p=52%. So 48% of the program is wasted on the overhead. In this calculation we used a large array size. We can use a small array size to obtain this portion again.

Using 10 cores, 8 threads gives best performance. Performance of 2 threads is closed to the performance of baseline.

On a 10 core CPU, we had the result as bellow:

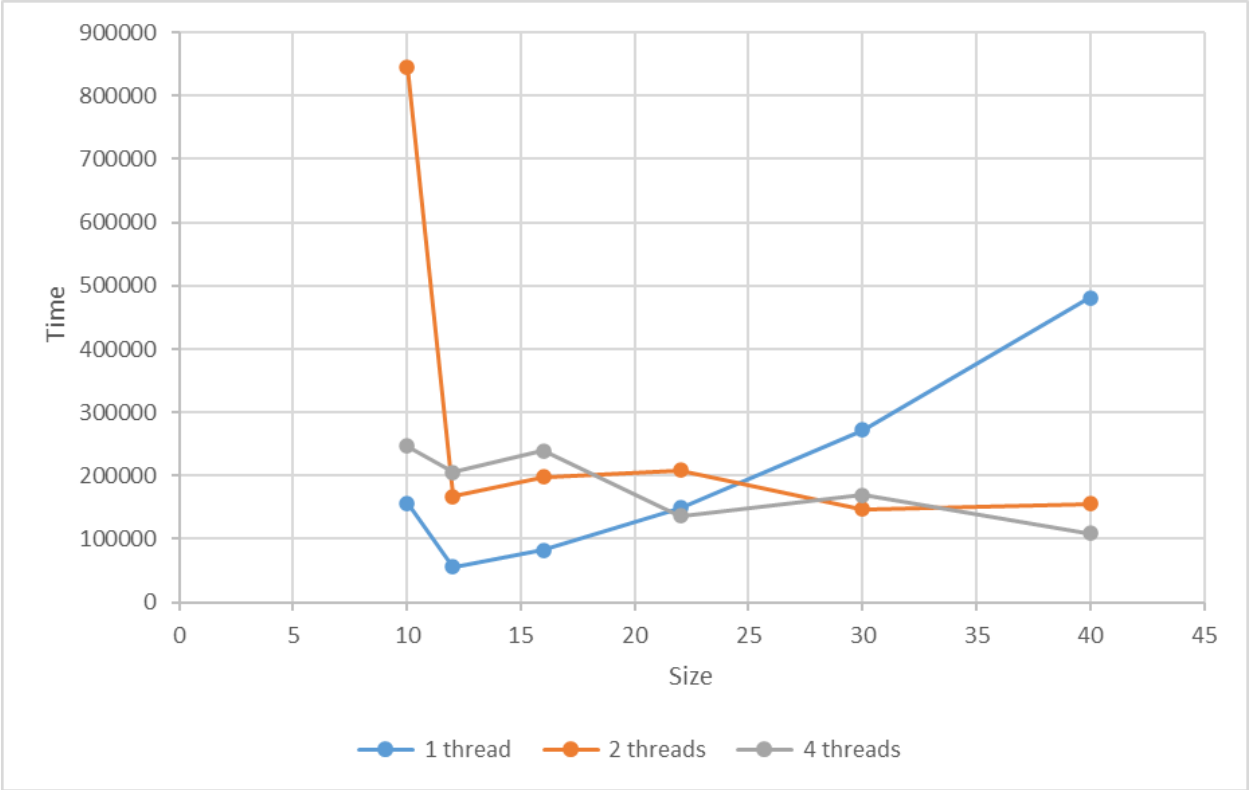




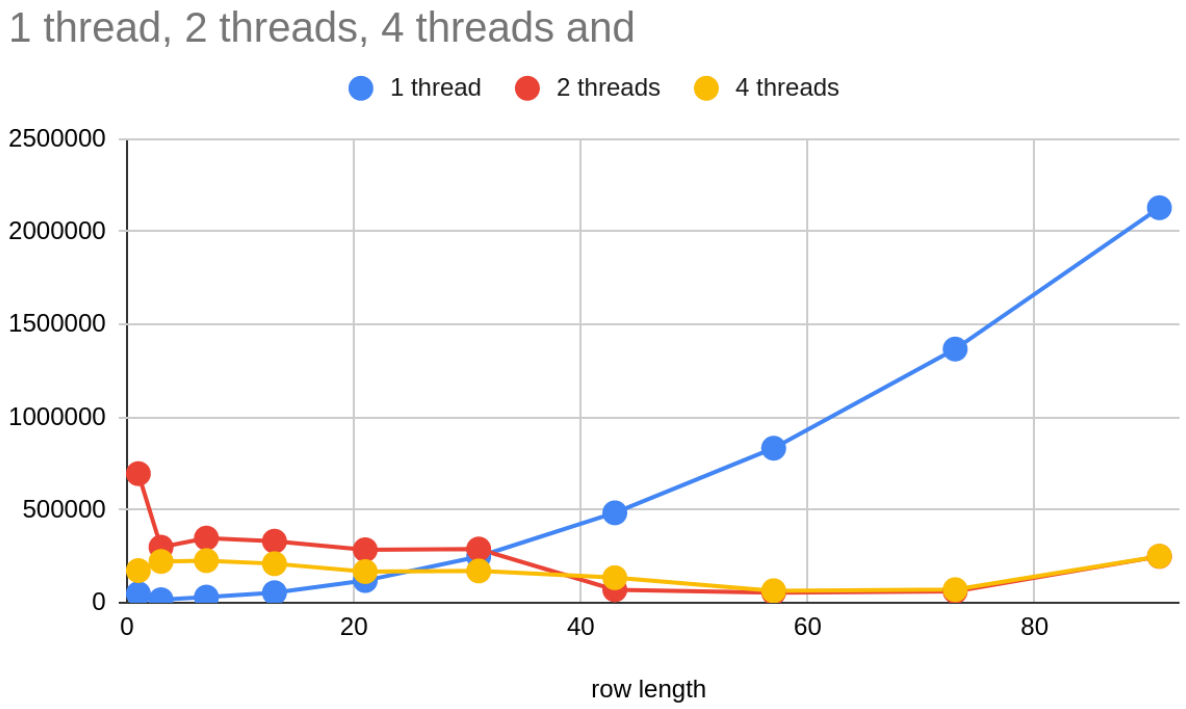
From Amdahl's law we now that this overall speedup must be $\frac{1}{(1-p)+p/s}$. Here s is 4/2=2 because we are using 4 threads. The overall speedup depends on the array size. For a large array size of 390 the overall speedup is: 1014650/601070 = 1.69. Thus, s = 54%. 54% of 1014650 is the overhead which is 547911. Here Overhead is for creating 2 threads so overhead of creating one is 273955.

For smaller array sizes we can say that everything is due to the overhead. For the array size of 12, the difference between 2 thread and 1 thread version is: 111289. Which is the overhead of creating a thread.

To clarify the difference more and see the break point we zoom out the graph:



We can see that somewhere at 20-25 the break point is happening between 1 thread and two others. Anyhow, we couldn't see significant breakpoints for 2 and 4 threads. We tried it on Lab machines, eng-grid, and local machines. There was always an anomaly. You may see the graph for lab machines too:



Task 4

For this part we write two different codes individually. The first version is only the simple pthread version of SOR and SOR_redblack and SOR_strip. (The first code is part4_v1.c and the second code is part4_v2.c). you can see the result for part4_v1.c below:

size	4 threads - SOR	SOR iters	red/black time	red/black iters	SOR_ji time	SOR_ji iters	SOR_blocked tir	SOR_blocked iters
32	2.02E+06	33	5.27E+06	118	7.28E+06	125	6.63E+06	125
56	6.06E+06	36	1.60E+07	118	2.26E+07	135	2.10E+07	135
96	2.11E+07	64	6.49E+07	320	8.02E+07	176	7.45E+07	176
152	1.04E+08	131	2.30E+08	469	4.83E+08	490	4.25E+08	490
224	3.40E+07	187	5.66E+08	536	1.40E+09	655	1.24E+09	655

We use pthread with 4 threads to implement SOR_OMEGA. Overall, It takes less time to finish the same size operations.

```
OMEGA = 1.90
OPTION=0 (normal serial SOR)
  iter 0 rowlen = 34
SOR() done after 32 iters
SOR() done after 33 iters
SOR() done after 31 iters
SOR() done after 33 iters
  iter 1 rowlen = 58
SOR() done after 35 iters
SOR() done after 35 iters
SOR() done after 36 iters
SOR() done after 36 iters
  iter 2 rowlen = 98
SOR() done after 63 iters
SOR() done after 64 iters
SOR() done after 64 iters
SOR() done after 64 iters
  iter 3 rowlen = 154
SOR() done after 131 iters
SOR() done after 131 iters
SOR() done after 131 iters
SOR() done after 131 iters
  iter 4 rowlen = 226
SOR() done after 186 iters
SOR() done after 187 iters
SOR() done after 187 iters
SOR() done after 187 iters
```

size	4 threads - SOR	SOR iters	4 thread - red/bl	red/black iters	SOR_ji time	SOR_ji iters	SOR_blocked tir	SOR_blocked ite
32	3.07E+06	0	2.80E+09	0	7.13E+06	125	6.69E+06	125
56	9.81E+06	0	1.88E+10	0	2.25E+07	135	2.13E+07	135
96	2.77E+07	0	2.14E+10	0	8.11E+07	176	7.56E+07	176
152	2.85E+07	0	2.31E+10	0	5.00E+08	490	4.40E+08	490
224	2.61E+08	0	2.32E+10	0	1.43E+09	655	1.26E+09	655

After we change the SOR_redblack to the 4 thread version, the SOR_redblack is fastest.

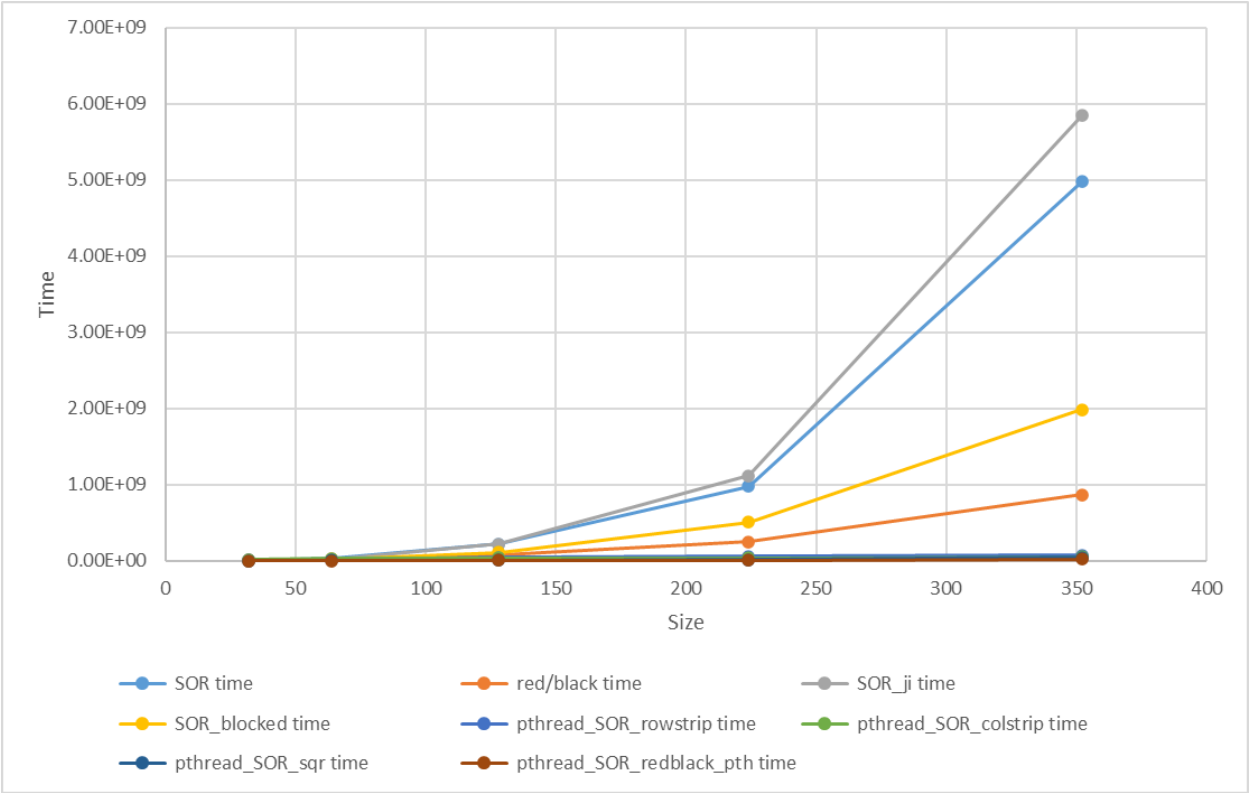
```
//define the number of strips and the size of the strip
int num_strips = 4;
int strip_size = rowlen / num_strips;

//define variable to keep track of the strips
int start, end;

while ((total_change/(double)(rowlen*rowlen)) > (double)TOL) {
    iters++;
    total_change = 0;

    //loop over each strip
    for (k = 0; k < num_strips; k++) {
        start = k * strip_size + 1;
        end = (k + 1) * strip_size + 1;
        for (i = start; i < end; i++) {
            for (j = 1; j < rowlen-1; j++) {
                change = data[i*rowlen+j] - .25 * (data[(i-1)*rowlen+j] +
                                                   data[(i+1)*rowlen+j] +
                                                   data[i*rowlen+j+1] +
                                                   data[i*rowlen+j-1]);
                data[i*rowlen+j] -= change * OMEGA;
                if (change < 0){
                    change = -change;
                }
                total_change += change;
            }
        }
```

size	SOR time	SOR iters	red/black time	red/black iters	SOR_ji time	SOR_ji iters	SOR_blocked tir	SOR_blocked ite	SOR_strip time	SOR_strip iters
32	3.44E+06	0	2.37E+06	0	7.15E+06	125	6.79E+06	125	6.54E+06	125
56	8.66E+06	0	3.44E+07	0	2.32E+07	135	2.19E+07	135	2.11E+07	135
96	1.83E+07	0	1.35E+08	0	8.56E+07	176	7.69E+07	176	7.46E+07	176
152	1.38E+08	0	1.41E+09	0	5.06E+08	490	4.42E+08	490	4.37E+08	490
224	2.36E+09	0	1.94E+09	0	1.44E+09	655	1.31E+09	655	1.27E+09	655

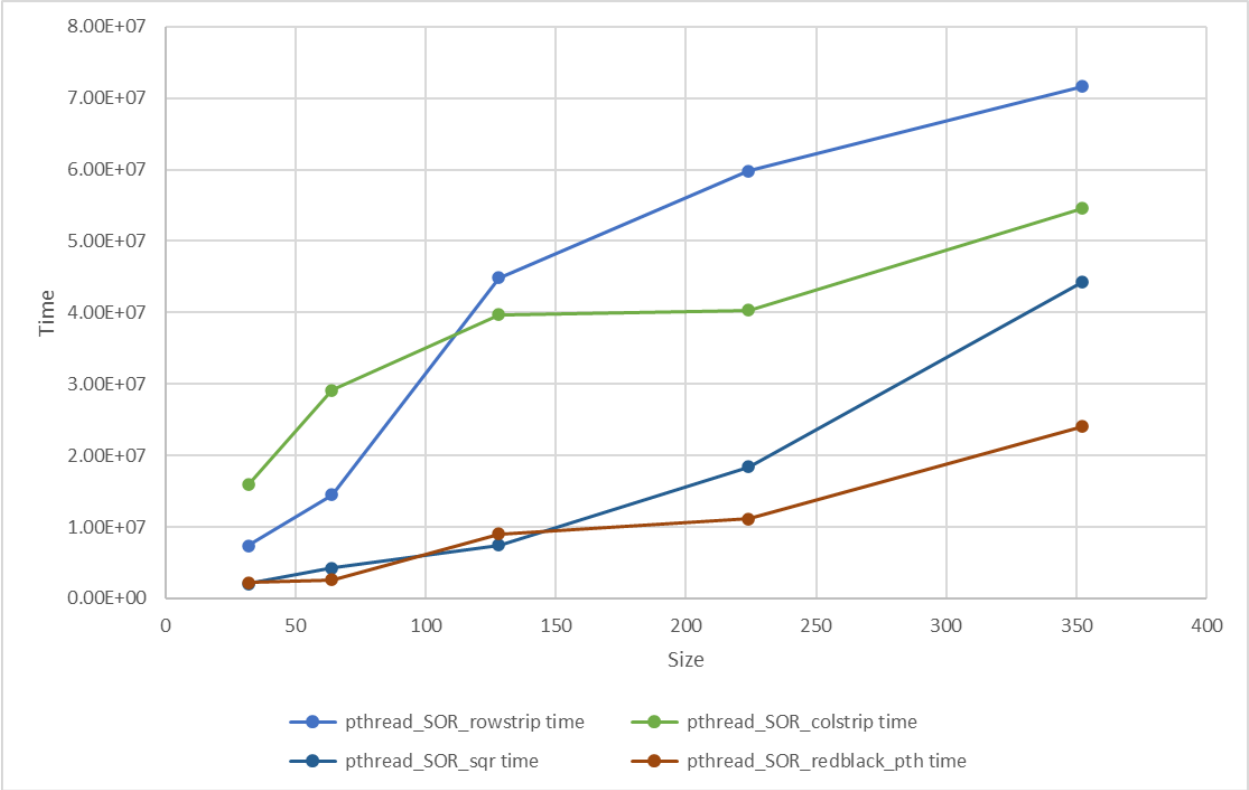
After we add the SOR_strips method for the decomposition of strips, we find the SOR_strips is faster than the SOR_blocked and SOR_ji. However it is slower than the pthread version of SOR.

On another machine we write a different piece of code that have 4 versions of pthread as described below:

We try this for 4 different kinds of pthread in terms of partitioning; row strip, column strip, square block and square block for red black method Their results are as below.

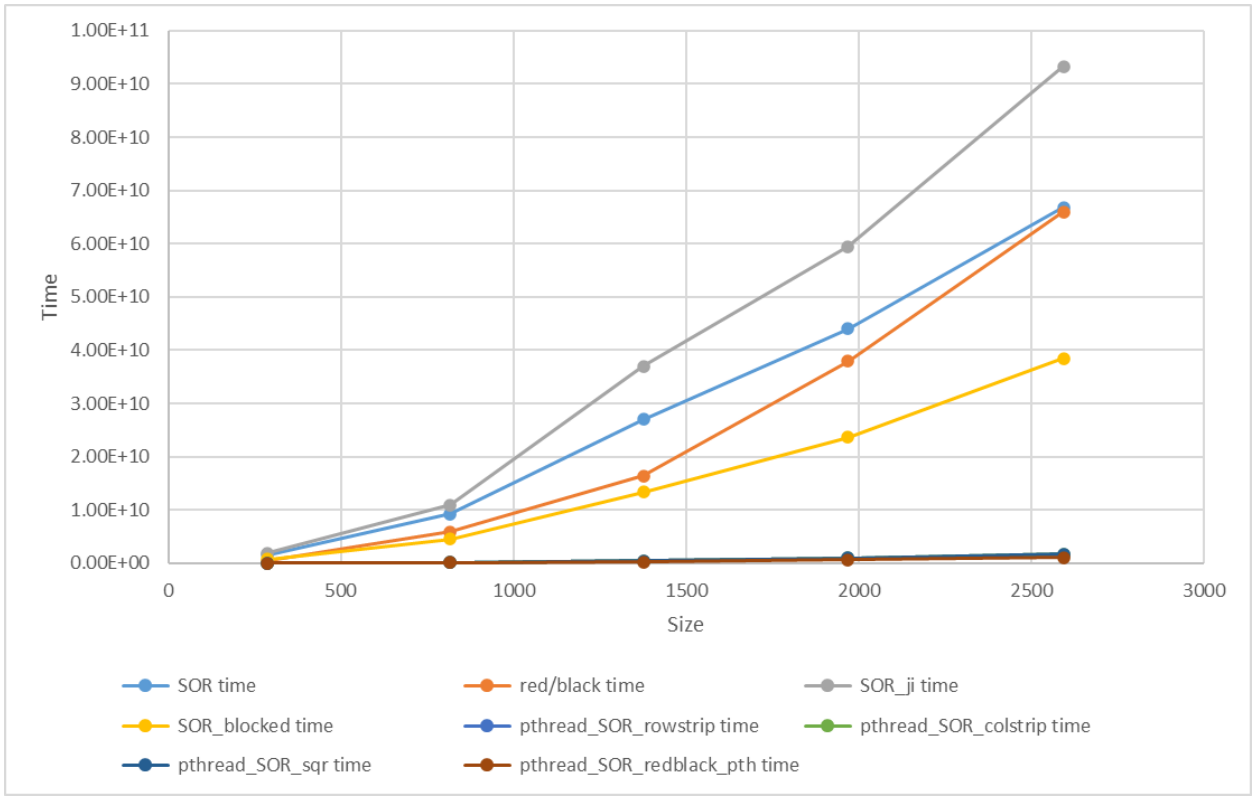


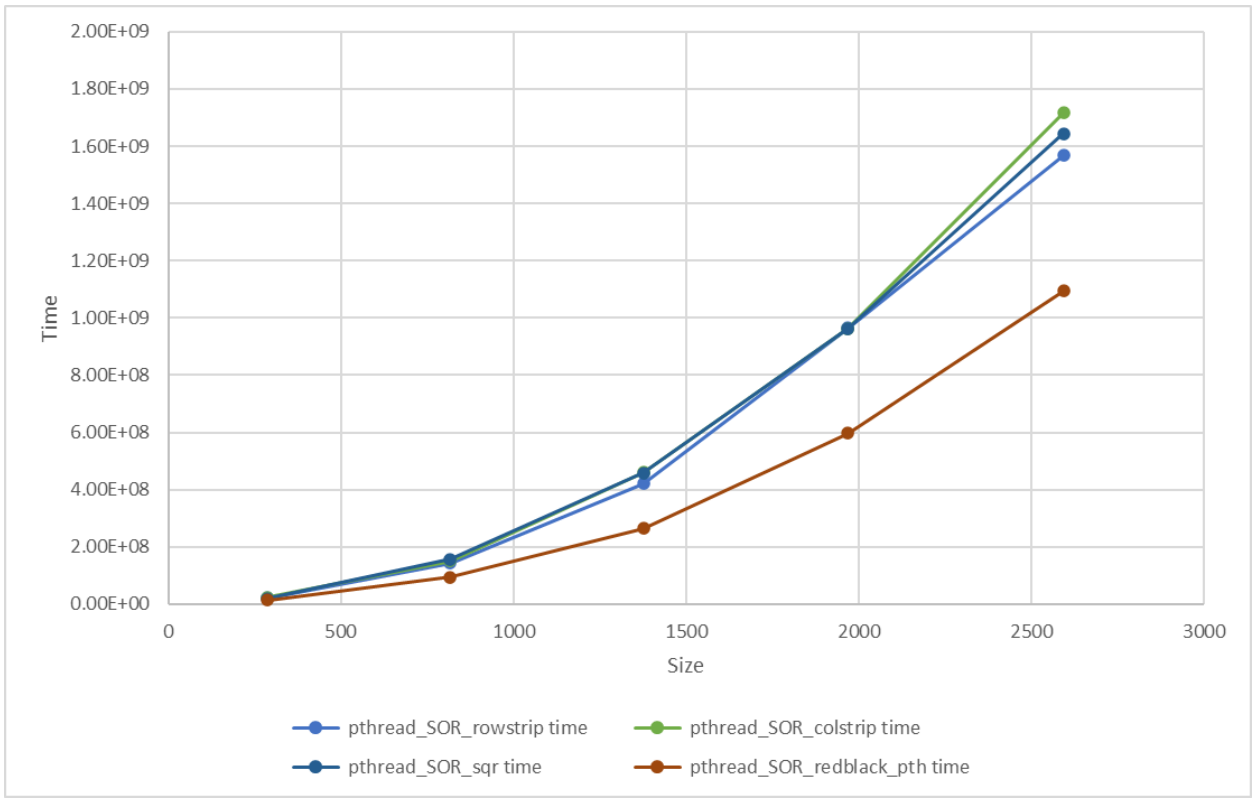
As we can see all of the threaded versions are better than others. To make a clear comparison between the threaded version in the next graph we omitted non-threaded versions.



As we can see the square versions are better than the two other versions, and the redblack one is better than the other.

As asked in the question we run the code for the versions that surpass the L3 cache. On the local machine we have a 24MB L3 cache. So a row length of >1732 would be larger than the cache size. In the below graphs the first 3 fitted to the L3, but the 2 others not.



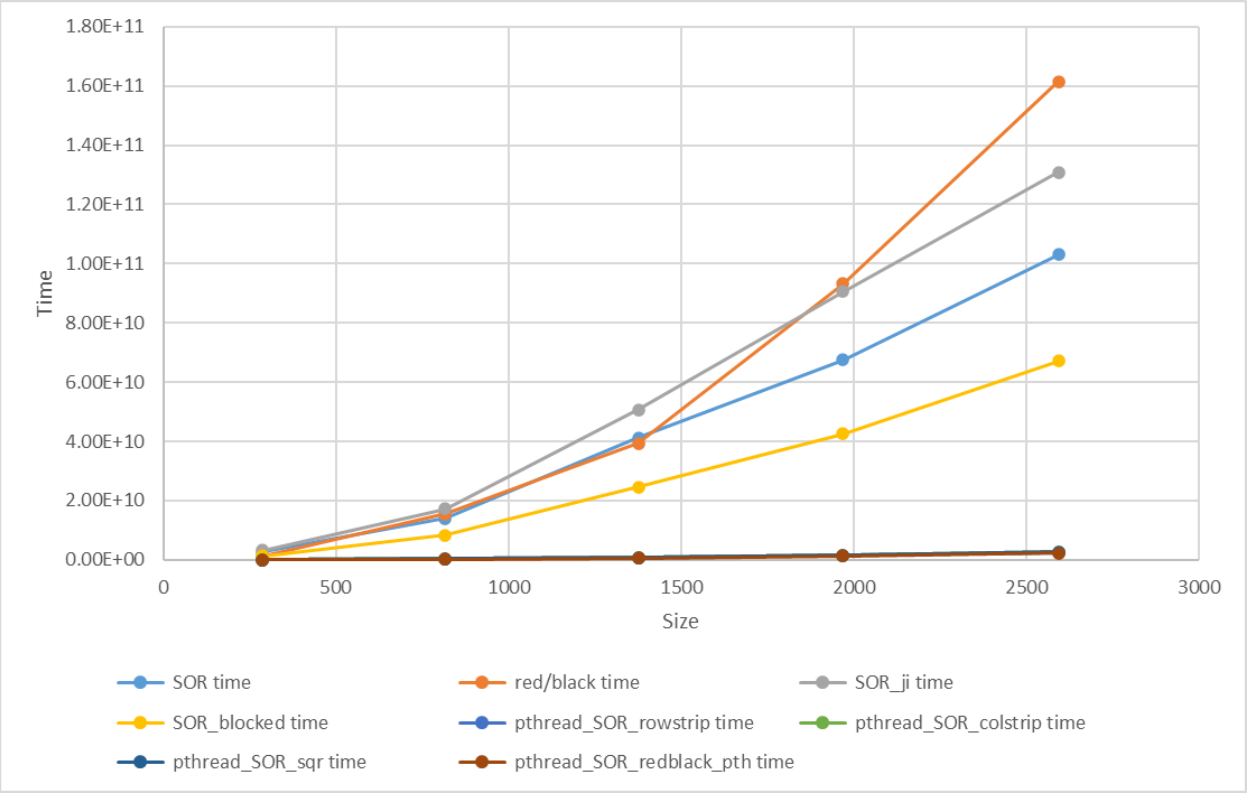


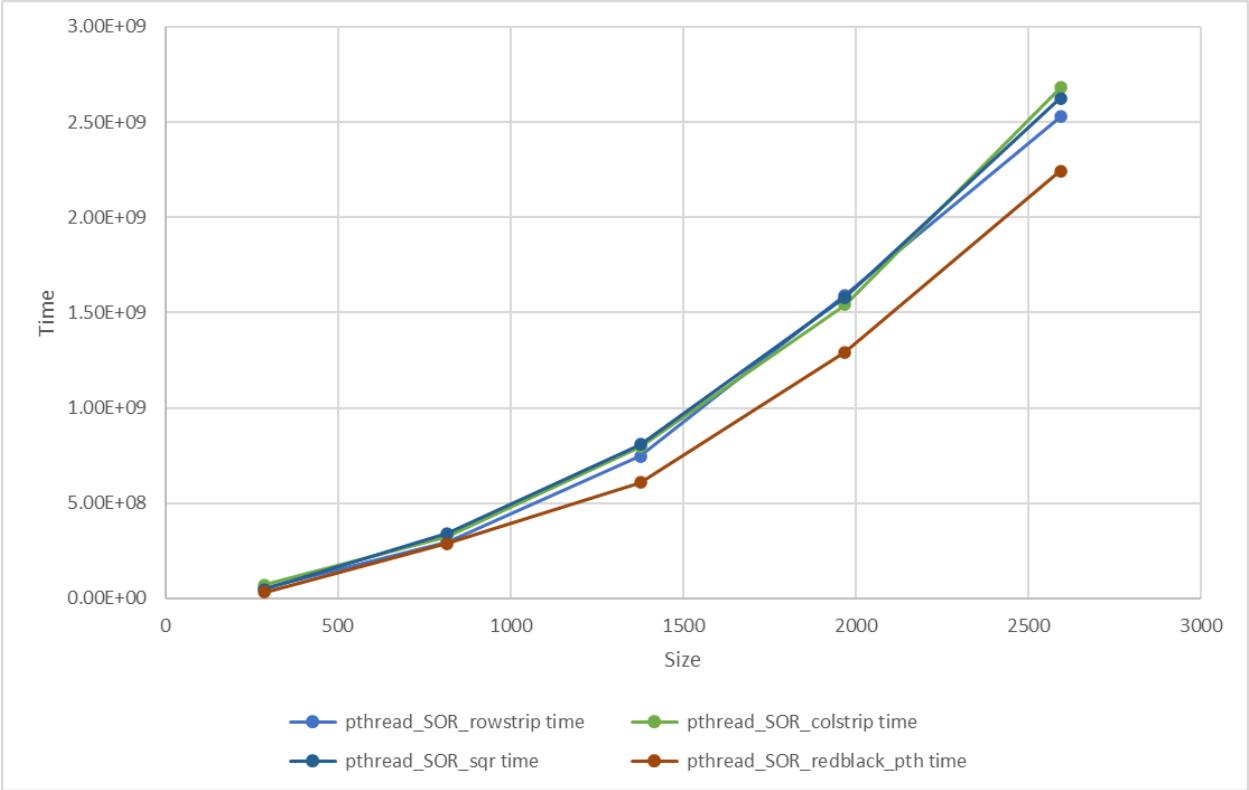
The table for this iterations are as below:

size	pthread_SOR_rowstrip time	pthread_SOR_rowstrip iters	pthread_SOR_colstrip time	pthread_SOR_colstrip iters	pthread_SOR_sqr time	pthread_SOR_sqr iters	pthread_SOR_redblack_pth time	pthread_SOR_redblack_pth iters
288	2.27E+07	597	2.48E+07	1027	1.95E+07	1383	1.39E+07	1486
816	1.43E+08	685	1.52E+08	1116	1.58E+08	1404	9.54E+07	1511
1376	4.20E+08	769	4.61E+08	1200	4.59E+08	1426	2.65E+08	1533
1968	9.66E+08	850	9.63E+08	1280	9.64E+08	1446	5.98E+08	1558
2592	1.57E+09	927	1.72E+09	1357	1.65E+09	1466	1.09E+09	1585

One observation that we had is that the number of iterations has increased significantly. This is because of the communication between processes. We didn't use any synchronization for when different threads try to read other boundaries. So it's possible that while the adjacent block boundaries value are changing another thread reads them. Since in the final result it doesn't make any difference we didn't take care of these dependencies. However, for updating the iterations we used mutex and calculated the average iterations for all threads.

On the lab machine we have a 10 core CPU with 25.6 MB L3 cache so the same size can be applied here too. In this case a row length of >1788 is larger than the cache size.





The result is pretty much the same as our local machine except for the anomaly that happens for the redblack version in the non-threaded version. It can be caused by other loads that were assigned to this CPU.

size	pthread_SOR_rowstrip time	pthread_SOR_rowstrip iters	pthread_SOR_colstrip time	pthread_SOR_colstrip iters	pthread_SOR_sqr time	pthread_SOR_sqr iters	pthread_SOR_redblack_pth time	pthread_SOR_redblack_pth iters
288	5.46E+07	597	7.10E+07	1027	4.91E+07	1383	3.39E+07	1486
816	2.96E+08	685	3.26E+08	1116	3.41E+08	1404	2.90E+08	1511
1376	7.47E+08	769	7.96E+08	1200	8.07E+08	1426	6.09E+08	1533
1968	1.59E+09	850	1.54E+09	1280	1.58E+09	1446	1.29E+09	1558
2592	2.53E+09	927	2.68E+09	1357	2.62E+09	1466	2.24E+09	1585