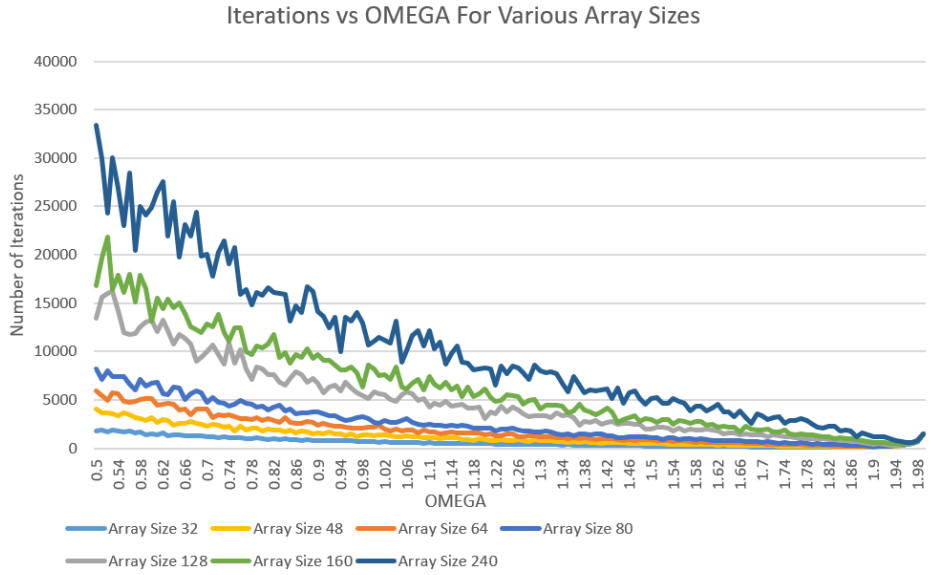
Lab 4 — High Performance Programming with Multicore and GPUs

Programming Assignment 5

Part1

As the L2 cache has 256K, and we want to set the array size to not fit in the L2, as the data type we used in the codes are double, also, as the double has 32 bits, so the largest array size that can fit in the L2 is $\sqrt{256 * 1024 \div 8} \approx 181$

Any array size larger than 90 will not fit in the L2 cache.
And the result is shown below:



We can see that from the overall variations of OMEGA from 0.5 to 2 with various array sizes, the number of iterations first decrease, and when it comes to an optimal value, in my graph is closed to 1.9, it is going to increase. So the shape of the graph looks like a U and its bottom is 1.9.
The array with a smaller size tends to iterate less than the larger ones.
Within the range of 0.5 and 1, OMEGA is found to be the most sensitive. I observed that when the OMEGA is less than 1, the number of iterations keep inconsistent with the OMEGA values.
For a given value of OMEGA, the number of iterations is also affected by the cache size. The number of misses increase if the array cannot fit in the cache and hence the number of iterations also increase.

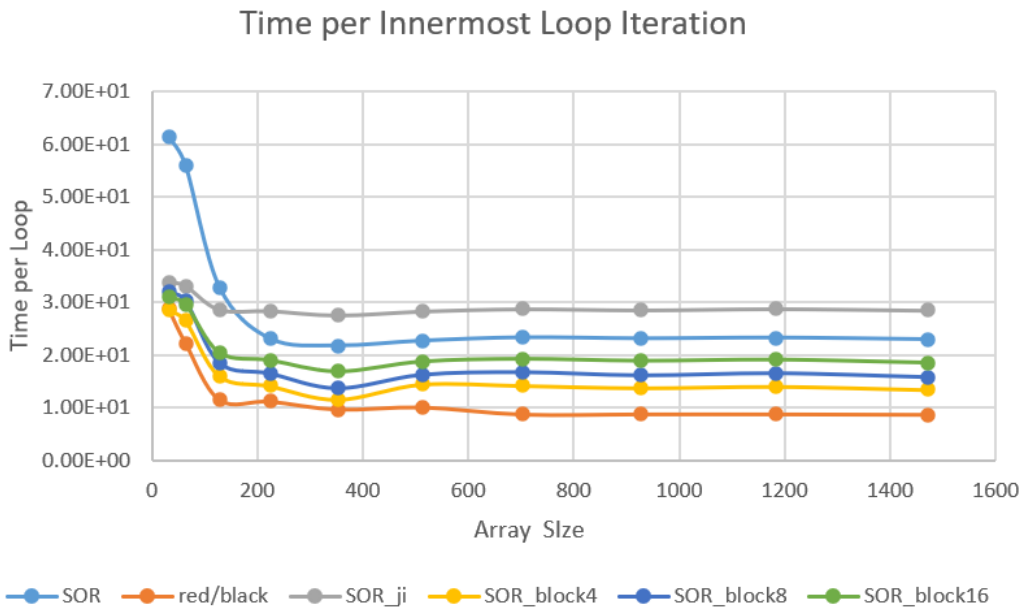
Part2

As the result in the part1 shows that the performance is best when OMEGA is closed to 1.9, we set the OMEGA in the test_SOR.c to be 1.90.

Also, as in the problem the array sizes are set to fits in L2 cache (less than 256KB) in the smallest but too big for L3 cache(12288KB), and the data type used is double, so we can set that the smallest and the largest row size for the array is

$\sqrt{256 * 1024 \div 8} \approx 181$ and $\sqrt{12288 * 1024 \div 8} \approx 1256$.

Also, all the numbers of the array size should be multiple of the block size, so we set the x, A, B and C to be 10, 8, 16 and 32, so that the smallest one of (GHOST + Ax^2 + Bx + C) is less than the L2 size and the largest is larger than L3 size.



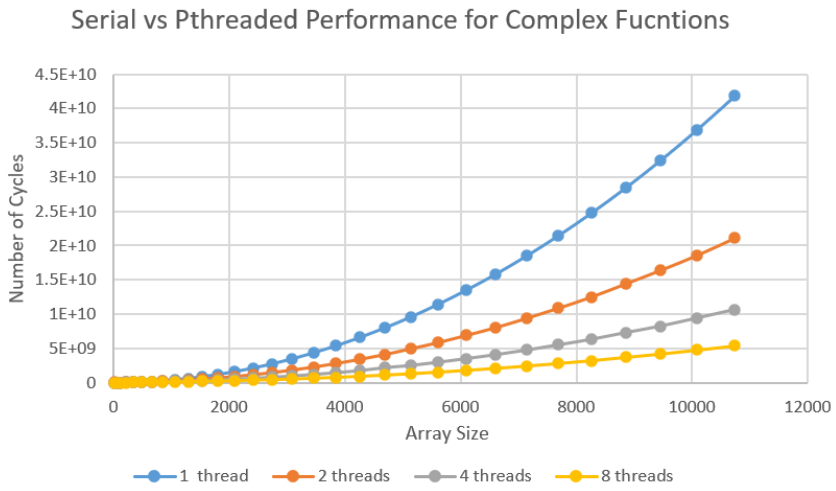
When the SOR is performed with red/black method, it performs the best. And then the block with size 4 also has a good performance, following by the block size of 8 and 16. The initial SOR method does the worst in the beginning, and then it performs better; and the reversed loop gives the worst.

The result is different from what I have thought in that the performance is becoming better with the increase of the block size.

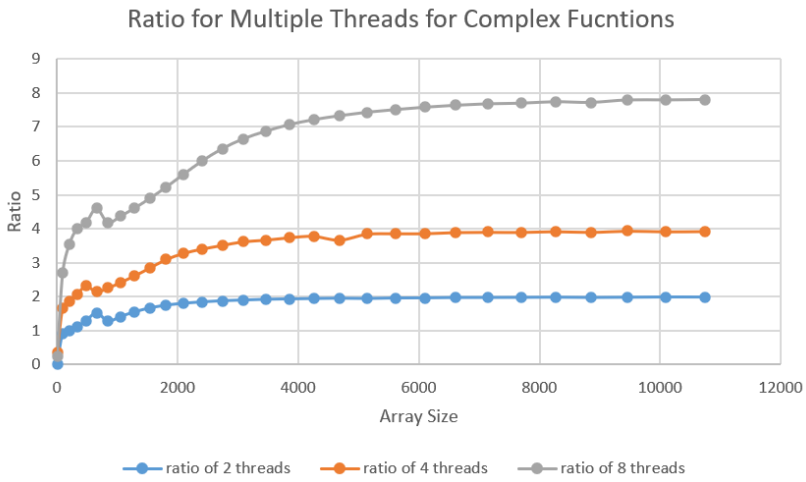
The possible reason for this could be the inability of the block to fit in the cache leading to more number of misses and hence a degradation in performance.

Part3

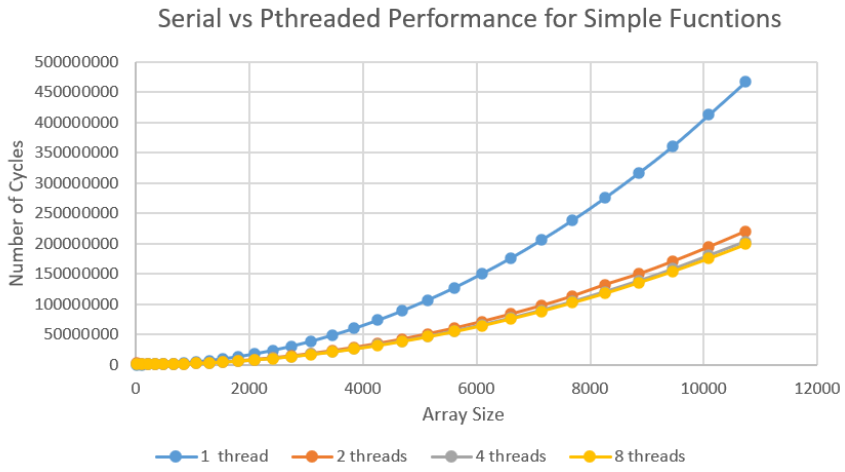
Given the result shown below, I found that when using the intensive function, the way using multithread performs much better than the sequential ones since the parallel in multi-threaded system can help running multiple tasks at the same time and thus increase performance.

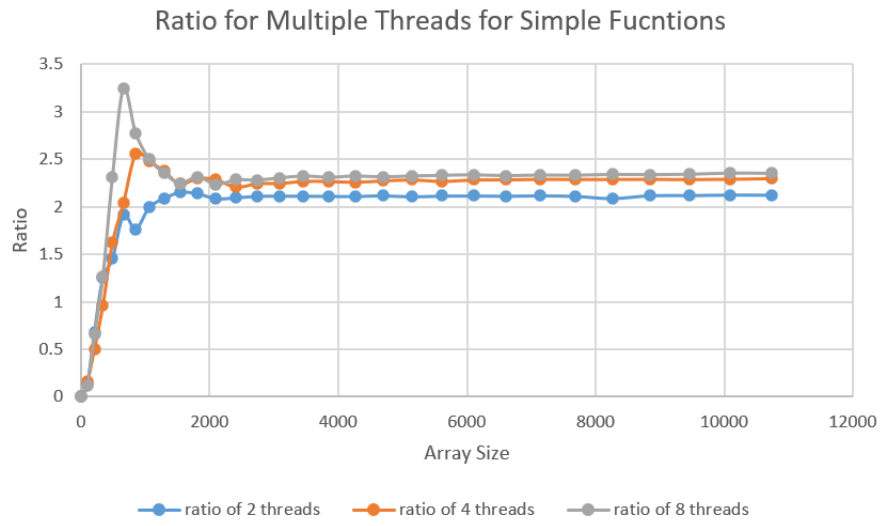


For making the ratio more easier to see, I made a graph showing the ratio for different number of threads. And we can see from the graph that when the array size goes beyond 6000, the ratio keeps the same.



When turning to the non-intensive function, the circumstance is much different. The one using one thread performs worst, but when using 4 threads or 8, their performance do not change much. Also, the ratio is not proportional as before. The reason behind that is probably because when calculating the simple functions, it does not require much calculating resources, and only if there are complex calculations can the all threads making themselves useful.





And for the break-even point, it requires that whenever using 2, 4 or 8 threads all them perform the same, and it turns out to be near, shown in the graph below. So when the array size is closed to 12 it is closed to the break-even point.

	1 thread	2 threads	4 threads	8 threads
10	92421	2362893	249628	171268
12	31800	335215	338215	269808
16	53616	248894	297672	248515
22	103478	300648	329964	266472
30	186033	268615	160500	268581
40	332858	424171	148267	252777
52	704388	483597	219504	261789
66	937351	683868	310903	295917
82	1512314	1004884	622776	494863
100	2193931	1300096	745454	648645

Part4

As it is requested in the problem that one array size should fit in the L3 cache while another does not, we can use the conclusion in part2: $\sqrt{12288 * 1024 \div 8} \approx 1256$.

So, we can set one array size to be 800 and another to be 1800.

For using different multithreaded versions of SOR, I tried decomposition by strips and nonadjacent strips. The following functions are added into the SOR_threaded_adj_rows() and SOR_threaded_nonadj_rows(). For the OMEGA, I used 1.9 as usual, and I used 4 threads. The result is shown below:

Size	SOR time	SOR iteration	SOR _adj_rows time	SOR_ adj_rows iteration	SOR_ nonadj_rows time	SOR_ nonadj_rows iteration
200+2	2.383e+08	295	5.334e+07	668	2.945e+08	3564
1800+2	3.557e+10	591	1.461e+10	3912	2.757e+10	6228

As the decomposition using adjacent rows can fit in smaller caches, it is faster when using same array size than the nonadjacent rows. Also, it is faster than the original SOR but needs more time to converge.

For making the result more easier to see, I calculated the innermost loop iteration as the same in the part 2, and here is the result:

Size	SOR	SOR_ adj_rows	SOR_ nonadj_rows
200+2	20.2	1.99	2.07
1800+2	18.6	1.15	1.37

When it turns to the circumstance that the array size is larger than the L3 cache, both adjacent and nonadjacent rows take a lot of time, but when calculating the overall efficiency it is much more efficient than the original SOR. The reason that it needs more iterations maybe because that when using multiple threads, it may lose some information from the border.