CSCI-GA.2945-002 : High Performance Computing **Homework:** 2
Date: 03/20/2022
Name: Yajur Ahuja **NetID:** ya2109

The link to the github repository is `https://github.com/yajurahuja/High-Performance-Computing`
I have written the make file for the linserv1.cims.nyu.edu server and if you are using that to compile, please first use a newer gcc version. You should run **module load gcc-9.2** before running the make command if running on the CIMS linserver.

2.1 Finding Memory bugs

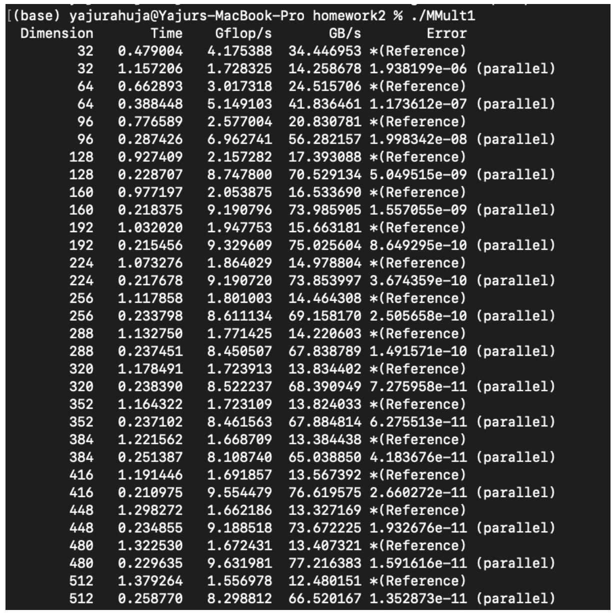
I have fixed the bugs and commented in the file itself. The programs compile with 0 valgrind errors.

2.2 Optimizing Matrix-matrix multiplication

Using MMult0 as a reference, implement MMult1 and try to rearrange loops to maximize performance. Measure performance for different loop arrangements and try to reason why you get the best performance for a particular order?
We want to order the loops to minimize the the number of cache misses. In that case it was best to order the the loop with the iterator p from 1 to k as the innermost loop. That way the number of accesses to an index in c is only mn. Total number of accesses are $2mn(k+1)$.
For my system mac Apple M1 Pro chip, I got the **best block size to be 8** . My laptop has 8 cores so it can use 8 threads simultaneously. I also tried and found improved using blocking and parallelism on the college CIMS linux server (linserver1.cims.nyu.edu).

2.2.1 Reference Solution

First we compare MMult0 with its own parallel versionl. The error is negligible. and we notice that the parallel version does better with larger matrix sizes. For smaller sizes, it may take a little more time due to the overhead of thread creation.
Below is that table displaying the Flop Rate and Bandwidth for different Matrix sizes.



```
(base) yajurahuja@Yajurs-MacBook-Pro homework2 % ./MMult1
 Dimension       Time    Gflop/s       GB/s        Error
        32   0.479004   4.175388  34.446953 *(Reference)
        32   1.157206   1.728325  14.258678 1.938199e-06 (parallel)
        64   0.662893   3.017318  24.515706 *(Reference)
        64   0.388448   5.149103  41.836461 1.173612e-07 (parallel)
        96   0.776589   2.577004  20.830781 *(Reference)
        96   0.287426   6.962741  56.282157 1.998342e-08 (parallel)
       128   0.927409   2.157282  17.393088 *(Reference)
       128   0.228707   8.747800  70.529134 5.049515e-09 (parallel)
       160   0.977197   2.053875  16.533690 *(Reference)
       160   0.218375   9.190796  73.985905 1.557055e-09 (parallel)
       192   1.032020   1.947753  15.663181 *(Reference)
       192   0.215456   9.329609  75.025604 8.649295e-10 (parallel)
       224   1.073276   1.864029  14.978804 *(Reference)
       224   0.217678   9.190720  73.853997 3.674359e-10 (parallel)
       256   1.117858   1.801003  14.464308 *(Reference)
       256   0.233798   8.611134  69.158170 2.505658e-10 (parallel)
       288   1.132750   1.771425  14.220603 *(Reference)
       288   0.237451   8.450507  67.838789 1.491571e-10 (parallel)
       320   1.178491   1.723913  13.834402 *(Reference)
       320   0.238390   8.522237  68.390949 7.275958e-11 (parallel)
       352   1.164322   1.723109  13.824033 *(Reference)
       352   0.237102   8.461563  67.884814 6.275513e-11 (parallel)
       384   1.221562   1.668709  13.384438 *(Reference)
       384   0.251387   8.108740  65.038850 4.183676e-11 (parallel)
       416   1.191446   1.691857  13.567392 *(Reference)
       416   0.210975   9.554479  76.619575 2.660272e-11 (parallel)
       448   1.298272   1.662186  13.327169 *(Reference)
       448   0.234855   9.188518  73.672225 1.932676e-11 (parallel)
       480   1.322530   1.672431  13.407321 *(Reference)
       480   0.229635   9.631981  77.216383 1.591616e-11 (parallel)
       512   1.379264   1.556978  12.480151 *(Reference)
       512   0.258770   8.298812  66.520167 1.352873e-11 (parallel)
```

Figure 2.2.1: Matrix Mult(MMult0) Vs Parallelized Matrix Mult(MMult0_p)

We see a 6 times speed up and around 9Gflops/s floprate just by parallellizing the matrix function Mmult0 without even blocking.

Now I will compare blocking max floprates to find out which is the best block size to choose.

Block Size	Max Floprate(Gflops/2)
8	7.0
12	6.7
16	5.6
20	5.6
32	4.6
64	4.6

We chose the block size of **8** as it was giving the best floprates. With the given block size and using Openmp to parallelize the block matrix multiplication, we get the following results. Below shows the table screen shot for the same. The reference refers to the MMult0 code which is the normal matrix multiplication code. The block refers to the MMult1_p which is the parallelized blocked matrix multiplication.

```
[(base) yajurahuja@Yajurs-MacBook-Pro homework2 % ./MMult1
 Dimension       Time    Gflop/s       GB/s        Error
         8   0.420853   4.752256  42.770300 *(Reference)
         8  38.673824   0.051715   0.103429 0.000000e+00 (block)
        56   0.589115   3.395375  27.648052 *(Reference)
        56   0.316696   6.316045   7.218337 1.813225e+02 (block)
       104   0.772991   2.587363  20.897931 *(Reference)
       104   0.151124  13.234220  14.252236 2.979619e+01 (block)
       152   0.949775   2.107584  16.971597 *(Reference)
       152   0.094938  21.084608  22.194325 3.429443e+01 (block)
       200   1.042655   1.933525  15.545545 *(Reference)
       200   0.084208  23.940718  24.898347 8.982157e+00 (block)
       248   1.086635   1.852871  14.882741 *(Reference)
       248   0.081355  24.748263  25.546594 1.273293e-11 (block)
       296   1.124200   1.799394  14.443781 *(Reference)
       296   0.075459  26.807647  27.532178 5.002221e-12 (block)
       344   1.133374   1.795858  14.408631 *(Reference)
       344   0.075269  27.041401  27.670271 5.456968e-12 (block)
       392   1.163046   1.760922  14.123317 *(Reference)
       392   0.076373  26.816202  27.363471 4.092726e-12 (block)
       440   1.175495   1.739196  13.945188 *(Reference)
       440   0.073950  27.645923  28.148576 4.092726e-12 (block)
       488   1.236827   1.691309  13.558200 *(Reference)
       488   0.074797  27.967123  28.425600 3.410605e-12 (block)
       536   1.271858   1.695055  13.585739 *(Reference)
       536   0.073840  29.196495  29.632263 2.160050e-12 (block)
       584   1.427911   1.673858  13.413795 *(Reference)
       584   0.083822  28.514238  28.904844 2.387424e-12 (block)
       632   1.190427   1.696440  13.592992 *(Reference)
       632   0.071390  28.288104  28.646181 2.501110e-12 (block)
       680   1.498756   1.678363  13.446646 *(Reference)
       680   0.092718  27.130180  27.449358 1.705303e-12 (block)
       728   1.390270   1.665123  13.339280 *(Reference)
       728   0.084318  27.455230  27.756936 1.818989e-12 (block)
       776   1.674010   1.674859  13.416142 *(Reference)
       776   0.080936  34.641340  34.998467 2.614797e-12 (block)
       824   1.344035   1.665064  13.336681 *(Reference)
       824   0.063170  35.426704  35.770653 1.136868e-12 (block)
       872   1.618702   1.638485  13.122914 *(Reference)
       872   0.073155  36.254793  36.587406 1.023182e-12 (block)
       920   1.902704   1.637013  13.110342 *(Reference)
       920   0.082800  37.617778  37.944889 9.663381e-13 (block)
       968   2.203625   1.646449  13.185202 *(Reference)
       968   0.104725  34.644611  34.930930 9.663381e-13 (block)
      1016   1.311827   1.598949  12.804180 *(Reference)
      1016   0.062284  33.677095  33.942269 6.536993e-13 (block)
      1064   1.478247   1.629701  13.049860 *(Reference)
      1064   0.064557  37.317414  37.597996 1.136868e-12 (block)
      1112   1.710961   1.607327  12.870180 *(Reference)
      1112   0.075799  36.281136  36.542152 1.023182e-12 (block)
```

Figure 2.2.2: Matrix Mult(MMult0) Vs Parallelized Blocked version Matrix Mult(MMult1_p)

824	1.344035	1.665064	13.336681	*(Reference)	
824	0.063170	35.426704	35.770653	1.136868e-12	(block)
872	1.618702	1.638485	13.122914	*(Reference)	
872	0.073155	36.254793	36.587406	1.023182e-12	(block)
920	1.902704	1.637013	13.110342	*(Reference)	
920	0.082800	37.617778	37.944889	9.663381e-13	(block)
968	2.203625	1.646449	13.185202	*(Reference)	
968	0.104725	34.644611	34.930930	9.663381e-13	(block)
1016	1.311827	1.598949	12.804180	*(Reference)	
1016	0.062284	33.677095	33.942269	6.536993e-13	(block)
1064	1.478247	1.629701	13.049860	*(Reference)	
1064	0.064557	37.317414	37.597996	1.136868e-12	(block)
1112	1.710961	1.607327	12.870180	*(Reference)	
1112	0.075799	36.281136	36.542152	1.023182e-12	(block)
1160	1.948821	1.601888	12.826148	*(Reference)	
1160	0.084445	36.968346	37.223300	1.307399e-12	(block)
1208	2.224165	1.585126	12.691506	*(Reference)	
1208	0.097551	36.140909	36.380253	1.193712e-12	(block)
1256	2.486486	1.593723	12.759937	*(Reference)	
1256	0.112196	35.320069	35.545037	1.477929e-12	(block)
1304	2.793250	1.587643	12.710887	*(Reference)	
1304	0.122364	36.241745	36.464087	1.534772e-12	(block)
1352	3.095600	1.596670	12.782810	*(Reference)	
1352	0.129597	38.138633	38.364305	1.364242e-12	(block)
1400	3.436758	1.596854	12.783955	*(Reference)	
1400	0.154242	35.580451	35.783768	1.705303e-12	(block)
1448	3.822256	1.588605	12.717616	*(Reference)	
1448	0.159692	38.023538	38.233613	1.591616e-12	(block)
1496	4.232825	1.581956	12.664109	*(Reference)	
1496	0.177035	37.823842	38.026109	1.875833e-12	(block)
1544	4.637680	1.587344	12.706978	*(Reference)	
1544	0.231607	31.784853	31.949541	1.818989e-12	(block)
1592	5.175074	1.559346	12.482608	*(Reference)	
1592	0.212201	38.028725	38.219824	1.705303e-12	(block)
1640	5.565296	1.585161	12.689017	*(Reference)	
1640	0.231363	38.130073	38.316073	2.103206e-12	(block)
1688	6.046718	1.590844	12.734292	*(Reference)	
1688	0.253428	37.957074	38.136965	2.103206e-12	(block)
1736	6.603816	1.584471	12.683067	*(Reference)	
1736	0.290731	35.990495	36.156350	2.046363e-12	(block)
1784	7.140900	1.590236	12.729020	*(Reference)	
1784	0.299415	37.926345	38.096418	2.103206e-12	(block)
1832	7.772547	1.582133	12.663975	*(Reference)	
1832	0.326115	37.708185	37.872850	2.387424e-12	(block)
1880	8.444531	1.573722	12.596472	*(Reference)	
1880	0.364353	36.473815	36.629023	2.046363e-12	(block)
1928	9.087418	1.577286	12.624837	*(Reference)	
1928	0.394596	36.324396	36.475120	2.557954e-12	(block)
1976	9.803628	1.573997	12.598351	*(Reference)	
1976	0.435689	35.417200	35.560589	2.501110e-12	(block)

Figure 2.2.3: Matrix Mult(MMult0) Vs Parallelized Blocked version Matrix Mult(MMult1_p)

We see that we get the highest flop rate of **38.1 Gflops/s**. Also for larger matrices the speedup compared to the reference implementation is around $24 - 25$ times which is a good speed up for a 8 thread system. It is due the blocking that the speed has increased more than the thread count.

Theoretical peak flop rate of my laptop Apple M1 pro :
Max clock speed for M1 pro processor: 3220 MHz
Number of Cores: 8 Cores
Max flop-rate: (Clock Speed) * (flops/cycle) * (cores) = 3220 MHz * (4) * 8 = 103 Gflops/s
Max Bandwidth: 204GB/s

Giving use around 37% performance percentage. (I am not sure which exact processor mac uses. I used the closest architecture processor mentioned on wikichips)

2.3 Finding OpenMP bugs

I have fixed the bugs and commented in the file itself.

2.4 OpenMP version of 2D Jacobi/Gauss-Seidel smoothing

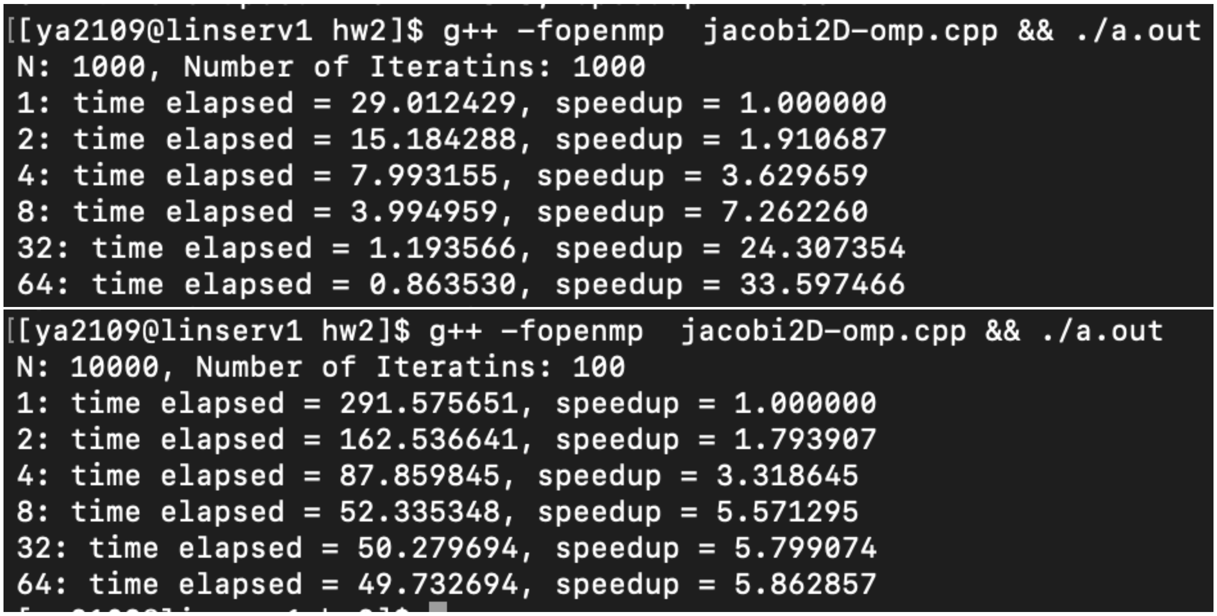
I first ran Jacobi smoothing on my laptop, Apple M1 pro mac which has a maximum of 8 threads. Below I have tabulated the results for Iterations = 1000. The results are quite proportional for different number of iterations also. I have done it for $N = 1000$ and $N = 10000$.

N = 1000		
Thread Count	Time taken(Sec)	Speedup
1	1.58	1
2	0.90	1.75
4	0.56	2.79
8	0.50	3.13

N = 10000		
Thread Count	Time taken(Sec)	Speedup
1	161.16	1
2	83.13	1.93
4	46.55	3.46
8	36.88	4.36

There is also overhead of having function calls and thread creation therefore the maximum theoretical efficiency is not achieved. But we see that with increase in the number of threads, there is increase in the speed up.

I also tried the algorithm on the CIMS Linux server 1. I have taken a screenshot of the results.



```
[[ya2109@linserv1 hw2]$ g++ -fopenmp  jacobi2D-omp.cpp && ./a.out
N: 1000, Number of Iteratins: 1000
1: time elapsed = 29.012429, speedup = 1.000000
2: time elapsed = 15.184288, speedup = 1.910687
4: time elapsed = 7.993155, speedup = 3.629659
8: time elapsed = 3.994959, speedup = 7.262260
32: time elapsed = 1.193566, speedup = 24.307354
64: time elapsed = 0.863530, speedup = 33.597466
[[ya2109@linserv1 hw2]$ g++ -fopenmp  jacobi2D-omp.cpp && ./a.out
N: 10000, Number of Iteratins: 100
1: time elapsed = 291.575651, speedup = 1.000000
2: time elapsed = 162.536641, speedup = 1.793907
4: time elapsed = 87.859845, speedup = 3.318645
8: time elapsed = 52.335348, speedup = 5.571295
32: time elapsed = 50.279694, speedup = 5.799074
64: time elapsed = 49.732694, speedup = 5.862857
```

Figure 2.4.4: Jacobi smoothing

Now doing the same with the gauss-seidel smoothing. The result on my laptop with Apple M1 pro with 8 thread. With number of iterations equal to 500.

N = 1000		
Thread Count	Time taken(Sec)	Speedup
1	0.97	1
2	0.58	1.67
4	0.39	2.47
8	0.28	3.46

N = 10000		
Thread Count	Time taken(Sec)	Speedup
1	102.64	1
2	54.13	1.89
4	32.27	3.18
8	27.9	3.66

Here also we see that with increase in number of threads, there is increase in the speed up.

And on the CIMS Linux server 1.

```
[[ya2109@linserv1 hw2]$ g++ -fopenmp  gs2D-omp.cpp && ./a.out
N: 1000, Number of Iteratins: 100
1: time elapsed = 2.901925, speedup = 1.000000
2: time elapsed = 1.619342, speedup = 1.792039
4: time elapsed = 0.815597, speedup = 3.558038
8: time elapsed = 0.445311, speedup = 6.516628
32: time elapsed = 0.205447, speedup = 14.124925
```

Figure 2.4.5: Gauss-Seidel smoothing

You can get the time and speedup for any value of N, iter, threads by changing the values in the code provided in the main function.