Homework 2

COEN 319
Haoyao Chen (ID: 1588503)

Cache size experiment

As the instructor states in a course announcement, the cache size experiment is hard to achieve in modern hardware. Therefore, I will skip this part for now. However, in creating the matrix multiplication algorithm, I will still vary the tile sizes and gauge their respective performance.

How to compile and run

Run
make
to compile and generate executable.

The executable takes 5 parameter values $1, $2, $3, $4, $5
$1 = Input A nrows
$2 = Input A ncols
$2 = Input B ncols2
$4 = tile size
$5 = thread number

In each execution, the program prints out the following comma separated values, they are Input A nrows, Input A ncols, INput B ncols2, combined data size, number of threads, tile size, execution time in microsecond.
Example:
1000, 1000, 1000, 2000000, 8, 256, 1638002

Run
sbatch matmult_omp.sh
to send job to the HPC. The shell script executes the matrix multiplication algorithm with different input sizes and different number of threads.

Parallelization Strategy

I used tiling to improve the algorithm's performance. My approach to tiling is based on the code in our OpenMP lecture. My program accepts a value for the tile size and breaks the input matrices into sub-matrices according to the tile size.

Upon the tiling algorithm, I used OpenMP for parallelization. I used the for statement and collapse statement to distribute work to worker threads. A certain number of worker threads (specified in the command line) is created, and each worker thread can work on an iteration of the triple for loop when it is idle.

In addition, I also made sure to make the update to the result matrix atomic to prevent contamination to the result.

An Analysis on the tile size's impact on performance

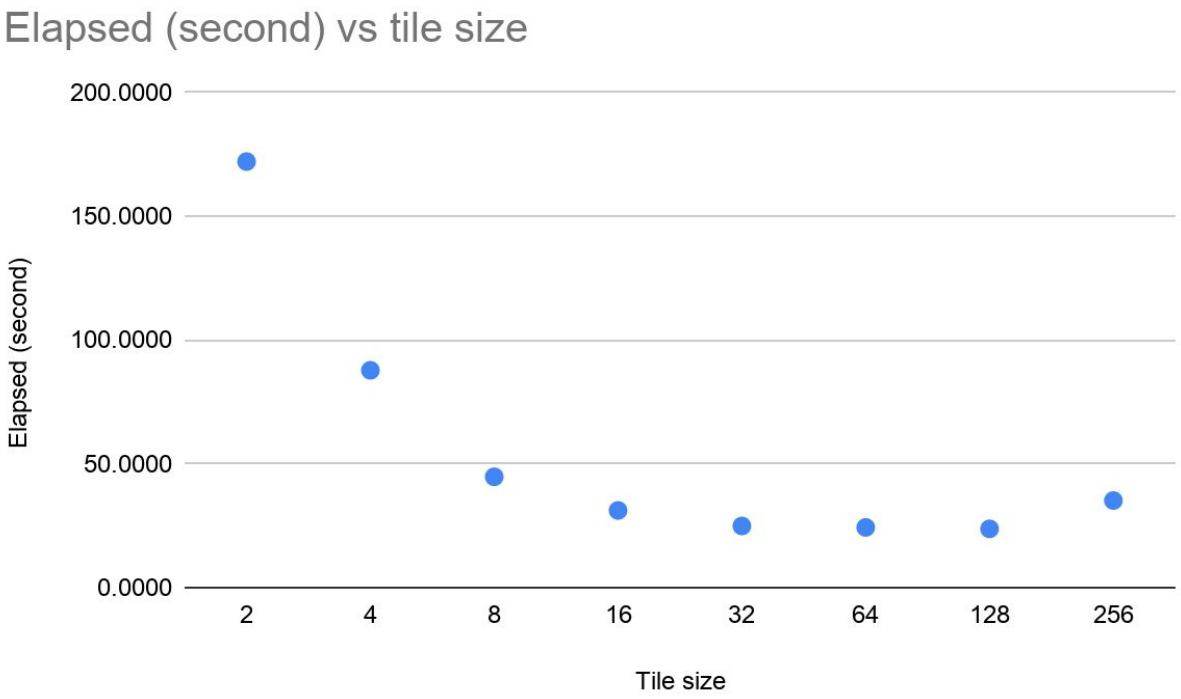
Using the algorithm described above, I set up an experiment to study the relationship between tile size and performance.

I used two 5000 x 5000 matrices as the input. I set the number of threads to 8. I then measure the performance using tile sizes 16, 32, 64, 128, 256, 512.

In another measurement, I recorded that the same input data takes more than 650 seconds to complete when we do not use tiling and parallelization. In this chart, we can see that the best case achieves the same calculation in just over 20 seconds, a more than 30x speedup.

As we can see in the chart below, there is a minor performance improvement from 16 to 32. However, beyond that, there is no longer any significant improvement in performance when we increase the tile size. Not only that, we see the performance decrease when we increase the tile size beyond 128.

The idea of tiling is to take advantage of data locality. I believe what this experiment shows is that tiling can improve performance if we choose our tile size right. If we choose a tile size that is too large, each thread would be responsible for too much data and we would no longer gain advantage from data locality.



Timing Results

In this table, I provide the timing results of the cases as required in the homework description. The tile size is fixed to be 32 as I find this setting to be optimal in prior experiments. I also provide the timing results of the serial version of the algorithm. In comparison, we can see that the parallelization achieved speed up from 3x to 34x depending on the different input sizes. We can see that in general, the larger the input size, the more speedup is achieved.

Case	A nrows	A ncols	B ncols	Thread #	Elapsed parallel (second)	Elapsed serial (second)	Speedup
1	1000	1000	1000	1	0.9607	3.1659	3.30
2	5000	5000	5000	1	129.2622	639.7257	4.95
3	1000	1000	2000	1	1.9067	6.0173	3.16
4	1000	2000	5000	1	10.2778	47.0888	4.58
5	9000	2500	3750	1	82.2176	546.6547	6.65
6	1000	1000	1000	2	0.4903	3.1659	6.46
7	5000	5000	5000	2	64.8944	639.7257	9.86
8	1000	1000	2000	2	0.9769	6.0173	6.16
9	1000	2000	5000	2	5.2745	47.0888	8.93

10	9000	2500	3750	2	40.9451	546.6547	13.35
11	1000	1000	1000	4	0.2718	3.1659	11.65
12	5000	5000	5000	4	34.0505	639.7257	18.79
13	1000	1000	2000	4	0.5222	6.0173	11.52
14	1000	2000	5000	4	2.8260	47.0888	16.66
15	9000	2500	3750	4	21.5993	546.6547	25.31
16	1000	1000	1000	8	0.1981	3.1659	15.98
17	5000	5000	5000	8	24.1889	639.7257	26.45
18	1000	1000	2000	8	0.3928	6.0173	15.32
19	1000	2000	5000	8	1.9612	47.0888	24.01
20	9000	2500	3750	8	15.6547	546.6547	34.92
21	1000	1000	1000	12	0.2035	3.1659	15.56
22	5000	5000	5000	12	24.4611	639.7257	26.15
23	1000	1000	2000	12	0.3925	6.0173	15.33
24	1000	2000	5000	12	1.9518	47.0888	24.13
25	9000	2500	3750	12	15.8558	546.6547	34.48
26	1000	1000	1000	14	0.2147	3.1659	14.74
27	5000	5000	5000	14	27.5666	639.7257	23.21
28	1000	1000	2000	14	0.4294	6.0173	14.01
29	1000	2000	5000	14	2.1930	47.0888	21.47
30	9000	2500	3750	14	17.9353	546.6547	30.48
31	1000	1000	1000	16	0.1993	3.1659	15.88
32	5000	5000	5000	16	24.5089	639.7257	26.10
33	1000	1000	2000	16	0.3894	6.0173	15.45
34	1000	2000	5000	16	1.8804	47.0888	25.04
35	9000	2500	3750	16	16.0711	546.6547	34.01
36	1000	1000	1000	20	0.1993	3.1659	15.88
37	5000	5000	5000	20	24.6986	639.7257	25.90
38	1000	1000	2000	20	0.3884	6.0173	15.49
39	1000	2000	5000	20	1.9819	47.0888	23.76
40	9000	2500	3750	20	16.1894	546.6547	33.77
41	1000	1000	1000	24	0.1989	3.1659	15.92
42	5000	5000	5000	24	24.8267	639.7257	25.77
43	1000	1000	2000	24	0.3888	6.0173	15.48

44	1000	2000	5000	24	1.9541	47.0888	24.10
45	9000	2500	3750	24	16.0663	546.6547	34.02
46	1000	1000	1000	28	0.2095	3.1659	15.11
47	5000	5000	5000	28	24.9494	639.7257	25.64
48	1000	1000	2000	28	0.3907	6.0173	15.40
49	1000	2000	5000	28	1.9779	47.0888	23.81
50	9000	2500	3750	28	16.2641	546.6547	33.61

Analysis

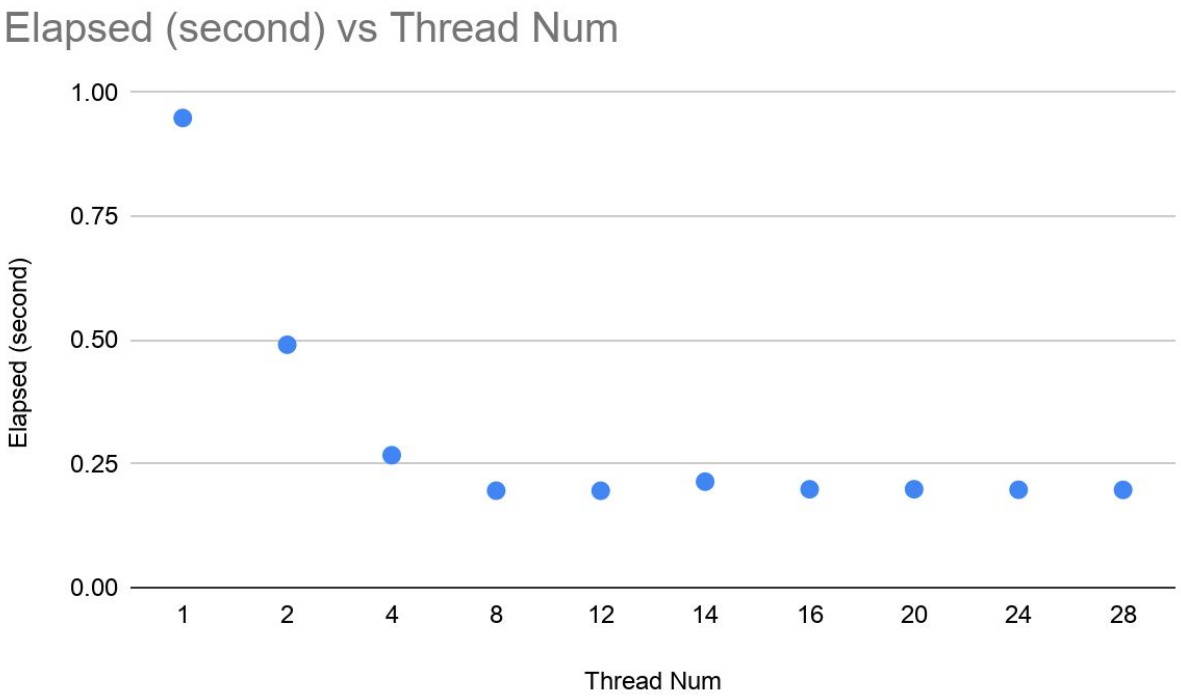
An Analysis on the number of threads' impact on performance

In this experiment, I use two 1000 x 1000 as input and set the tile size to 32. I vary the number of threads from 1 to 28.

As we can see in the chart below, the performance gain is dramatic from thread_num=1 to thread_num=8. I believe that the performance gain here is due to the utilization of parallelization on a multi-threaded, high-performance system.

However, we can also see that the performance begins plateau when the number of threads exceeds 8. I believe this is because the cost in synchronization begins to overweigh the performance gain from parallelization.

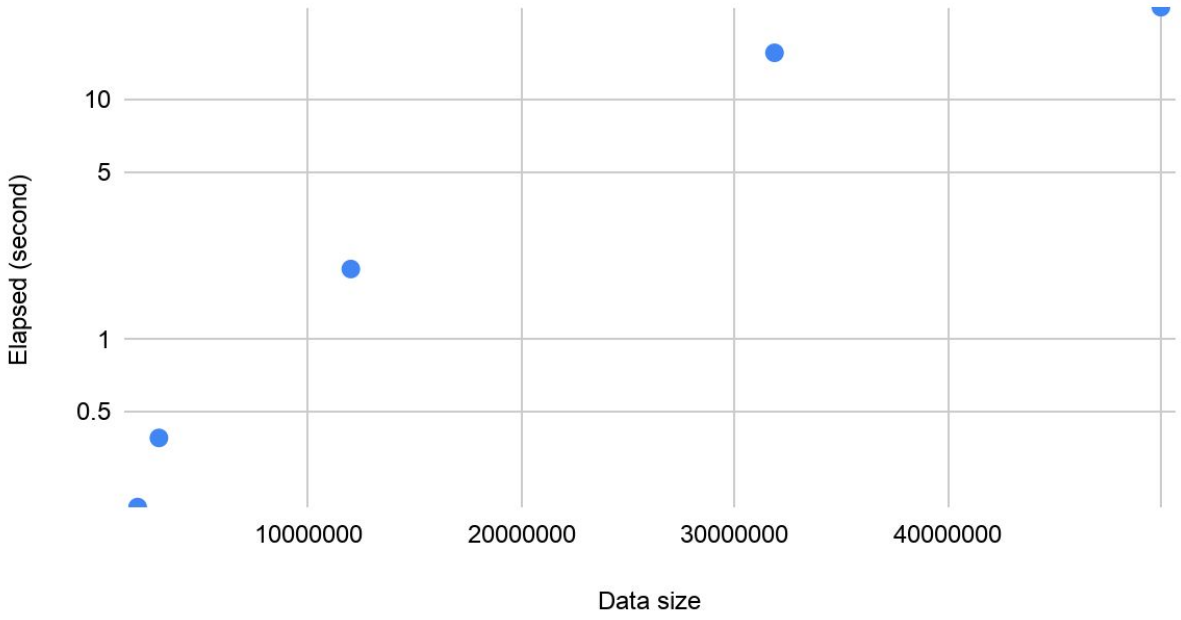
From this measurement, we can see that the algorithm's scalability is limited. That is, the performance cannot keep growing in proportion to the number of threads.



An Analysis on the input data size's impact on performance

In this experiment, I measure the performance of the matrix multiplication algorithm using different input sizes. The number of threads is fixed at 8 and the tile size is fixed at 32. This is because I find those settings to be optimal based on prior experiments.

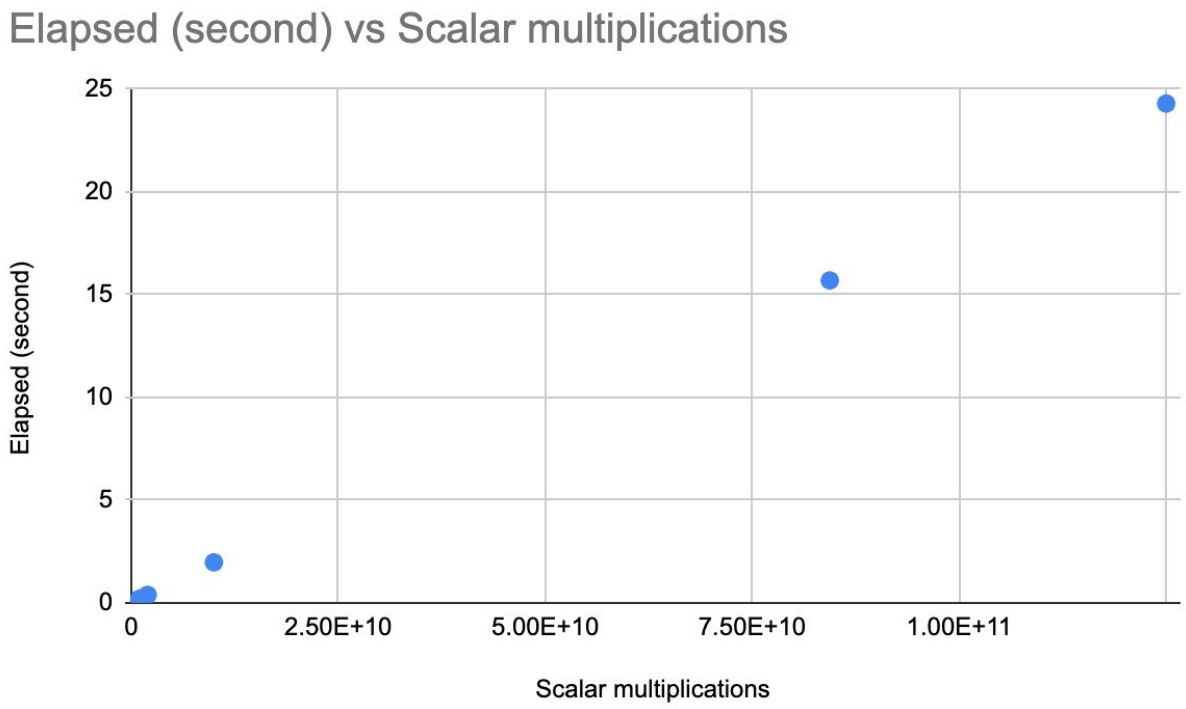
The data sizes in the below chart refer to the combined size of the two input matrices. The vertical axis shows the execution time in a log scale. We can see that the execution time grows nearly exponentially as the data size grows. I believe this is because the matrix multiplication algorithm requires a triple for loop. The nature of the calculation means that the run time would increase exponentially relative to the data size.



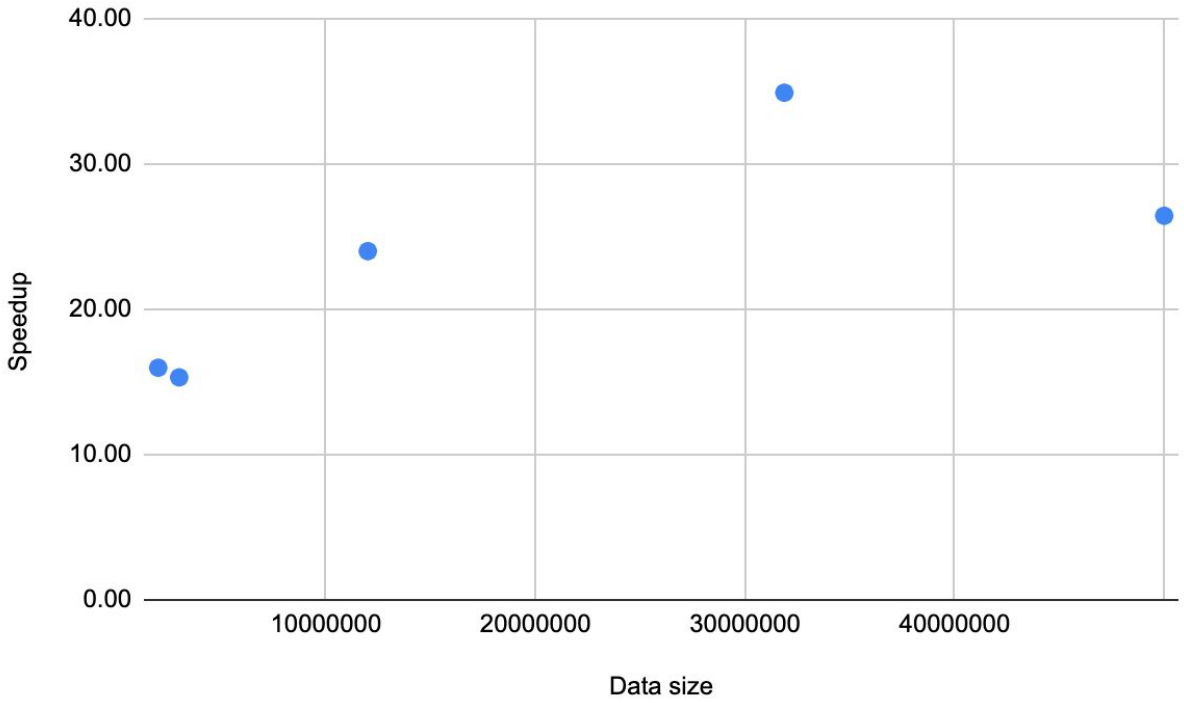
We should also note that the amount of work to be done in the algorithm not only relates to the total input size, but also relates to the shapes of the input matrices. This is because the shapes of the input matrices determine the number of scalar multiplications to be done.

The number of scalar multiplications to be done is given by
= A nrows x A ncols x B ncols2

In the chart below, we can see that the execution time is linearly correlated with the number of scalar multiplications.



What is also noting is that speedup grows with data size. In the chart below, I show the performance gain at different data sizes. Overall, when we parallelize an algorithm, the larger the data size is, the more performance gain we can achieve.


40.00
30.00
20.00
10.00
0.00
Speedup
10000000
20000000
30000000
40000000
Data size