Homework 2

Sajjad Azami - 9231031
Special Topics: Multicore Programming

April 26, 2018

Preprocessing. In order to be able to use OpenMP, we first include the library. Next, we should measure the time of the *multiply()* function. This can be done using *omp_get_wtime()* function. Given that we want to test our results for various number of cores, each run consists of an iteration over *num_of_threads*. Note that we fix *#pragma omp parallel* for the most inner loop in order to obtain better results. For the other two settings, we will parallelize the first and second loops. A function named *multiply_validator()* has been added to the code in order to roughly validate the results of *multiply()* function before and after parallelization. For each matrix size, the dimension of the matrix can be calculated by $\sqrt{s/4}$ where s is the size of the matrix in bytes. The rest of the report consists of the results for the two settings and a section for conclusion.

1D. For the first setting(1D), we will parallelize matrix A's row iteration. Figure 1 shows an illustration of the method. Final results are averaged over 5 runs and have been reported in Table 1.

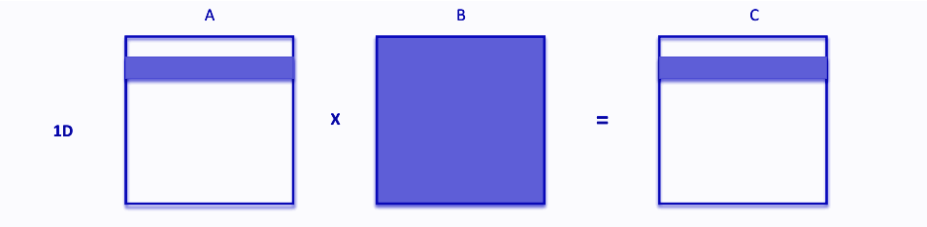

Figure 1: Illustration of the 1D setting.

Number of Threads	Matrix Size			
	1MB	4MB	16MB	64MB
1	0.747967	6.582945	43.033203	431.723954
2	**0.465373**	**3.515298**	27.929443	313.056427
4	0.579131	4.229820	**26.119875**	**213.867240**
8	0.981710	5.543369	27.500152	220.764496

Table 1: Summary of running times for 1D parallelization. Each output has been averaged over 5 runs.

For each matrix size and number of threads, we then calculate the speed-up by $s_k = time_1/time_k$ where k is the number of threads. Results are summarized in Table 2.

Number of Threads	Matrix Size			
	1MB	4MB	16MB	64MB
1	-	-	-	-
2	1.6072419328	1.8726563153	1.5407827145	1.3790611429
4	1.2915333491	1.5563179993	1.6475271417	2.0186539743
8	0.7619021911	1.1875350531	1.5648350962	1.9555859834

Table 2: Speed-up ratio for 1D parallelization.

2D. For the second part(2D), we will parallelize matrix A's row iteration. Figure 2 shows an illustration of the method and final results have been reported in Table 3.

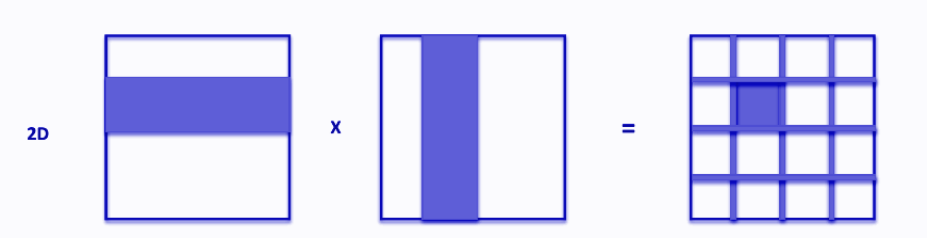

Figure 2: Illustration of the 2D setting.

Number of Threads	Matrix Size			
	1MB	4MB	16MB	64MB
1	0.677615	6.632248	46.782576	441.957828
2	0.423074	**3.431854**	23.027644	280.445581
4	0.402491	3.659250	22.976554	221.230655
8	**0.383583**	3.654259	**22.839362**	**212.738062**

Table 3: Summary of running times for 2D parallelization. Each output has been averaged over 5 runs.

Again, for each matrix size and number of threads, we calculate speed-up. Results are summarized in Table 4.

Number of Threads	Matrix Size			
	1MB	4MB	16MB	64MB
1	-	-	-	-
2	1.6072419328	1.8726563153	1.5407827145	1.3790611429
4	1.2915333491	1.5563179993	1.6475271417	2.0186539743
8	0.7619021911	1.1875350531	1.5648350962	1.9555859834

Table 4: Speed-up ratio for 2D parallelization.

Conclusion. Based on the results, we can infer that for sufficient amount of data, increasing the number of threads will improve the performance of the *multiply* function. Meanwhile, in all cases, 8 threads do not perform better than 4 and the reason this happens can be the increasing over-head of the threading procedure. It is evident that for small amounts of data, even 4 number of threads drops the performance compared to 2. We also can observe that the 2D setting outperforms the 1D in their best performances. This was somehow expected even before running the experiments, given that the 2D parallelizes the function better.