HW2 Gaussian Elimination in parallel

Yedong Liu

A20344124

Algorithm

Well, there is a very obvious but also naive approach to solve the problem which I guess almost every one can find. Look at the original serial approach, the normalization line is the outer loop and will increase one in every loop cycle, and within the outer loop, for the row loop, it is easy to find out that different rows calculate independently from each other under the same normalization line. The only parameter is the multiplier which can be calculated by the data of row[row][norm] and row[norm][norm], and this is completely independent from calculations in other rows. So I started my parallel programming here. I did not intend to parallel from the outer loop, but under every normalization line, I will assign each line to different processes or threads sequentially according to their rank or thread id. For example, if we have 4 threads, 10 rows and the current normalization line is row[0], I will assign row[1], row[5] and row[9] to p_0, row[2] and row[6] to p_1, row[3] and row[7] to p_2, row[4] and row[8] to p_3. Each thread will do the calculation and work independently under the same normalization line. After all the rows have been updated, the normalization line will go down, and all the threads will start working again and updating the new rows until the last row has been updated.

For the pthread program, the main idea about my algorithm has been discussed above, every thread is in charge of one row under a certain normalization line. After one outer loop cycle completed, the normalization line goes down to the next row, and the threads from 0 to n are assigned to new lines from row[norm+1] to row[N-1] to do calculation.

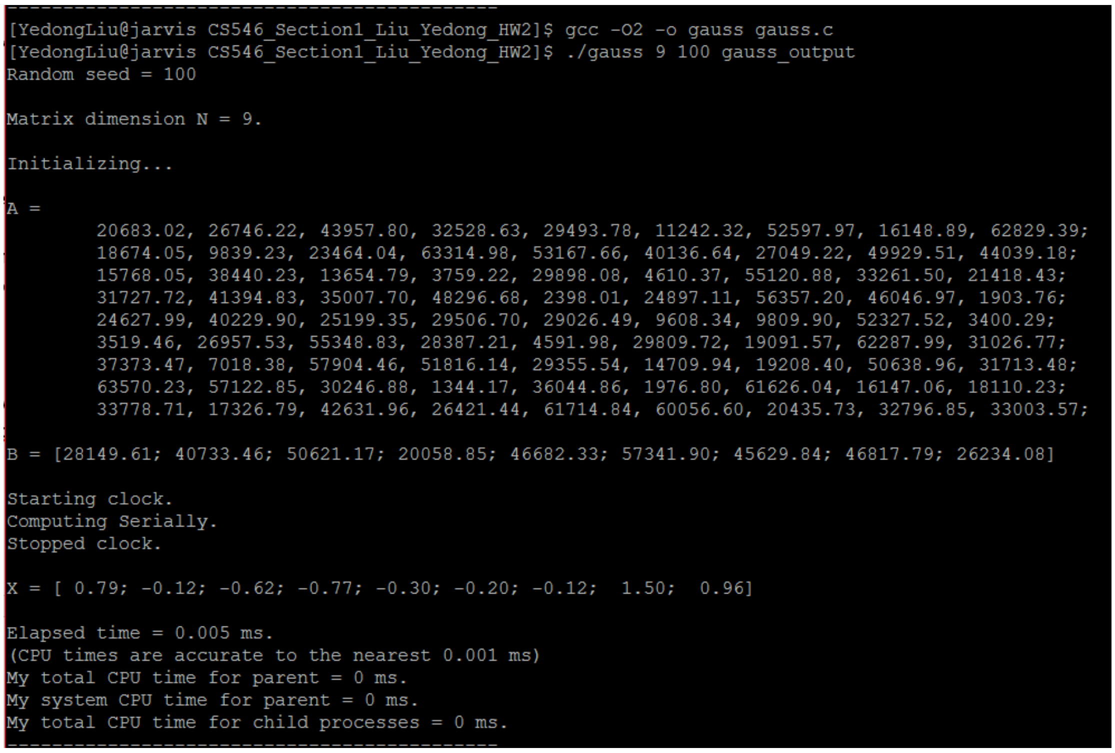
For the MPI program, however, the algorithm is the same and the most part of the parallel share the same idea but there are still some differences. Since the mpi processes do not share the same memory, we need to communications between processes. My approach is that the rank 0 process will initiate both matrices and store all data while others do not. Then rank 0 will send each specific row data to each process according to their rank. Similar to pthread, assume we have 4 processes and 10 rows and the current normalization line is row[0], I will assign row[1], row[5] and row[9] to p_0, row[2] and row[6] to p_1, row[3] and row[7] to p_2, row[4] and row[8] to p_3. After every row[norm+1] completed its calculation, the data will be sent back to rank 0 if the process' rank is not 0, meanwhile rank 0 will also do its own work on its own rows, and this completes the update.

Correctness

In the original serial code, the matrix A and matrix B will only be printed if N<10, so obviously we can use the serial code and a constant seed=100 to check our correctness. If the output matrix A and matrix B from the serial code are the same with the matrices from the pthread or MPI code, we constructed the same matrices, and if the output of matrix x is the same with the matrix from pthread or MPI code, we clearly have the right answer.

Note: I modified the gauss.c file, adding an output file name as an argument in the command line.

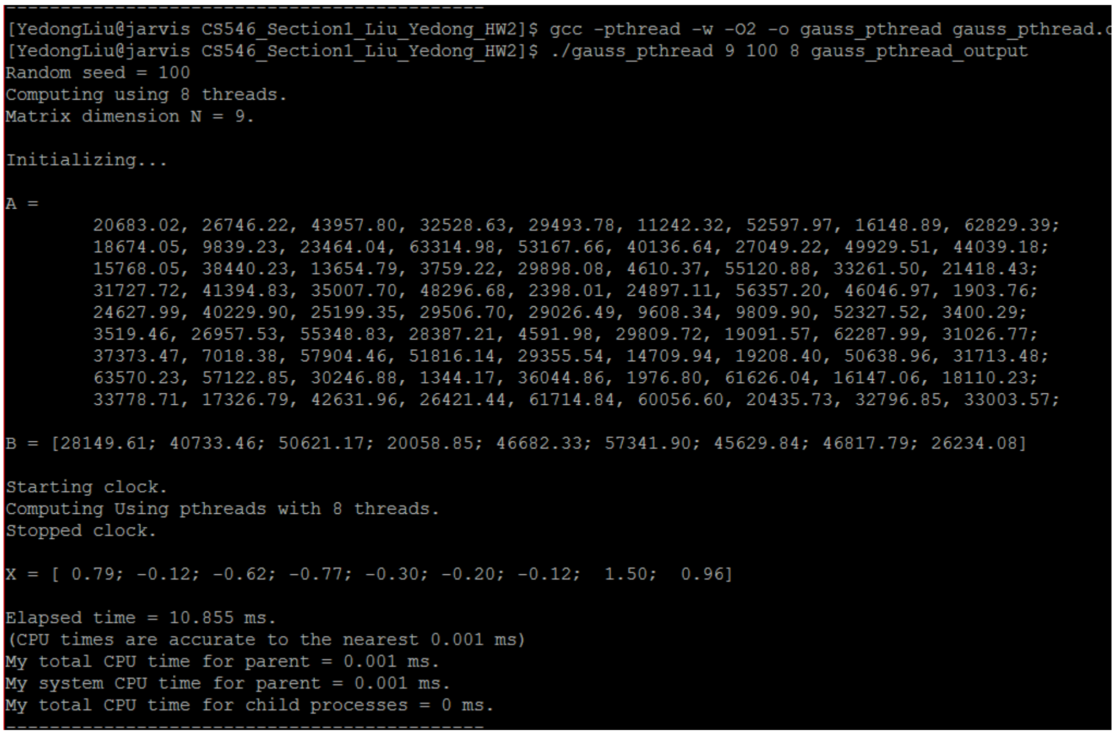
So, for the serial code, we compile with "gcc -O2 -o gauss gauss.c", and execute with " ./gauss 9 100 gauss_output", and the result is the following Figure 1:



```
----------------------------------------------
[YedongLiu@jarvis CS546_Section1_Liu_Yedong_HW2]$ gcc -O2 -o gauss gauss.c
[YedongLiu@jarvis CS546_Section1_Liu_Yedong_HW2]$ ./gauss 9 100 gauss_output
Random seed = 100

Matrix dimension N = 9.

Initializing...

A =
	20683.02, 26746.22, 43957.80, 32528.63, 29493.78, 11242.32, 52597.97, 16148.89, 62829.39;
	18674.05, 9839.23, 23464.04, 63314.98, 53167.66, 40136.64, 27049.22, 49929.51, 44039.18;
	15768.05, 38440.23, 13654.79, 3759.22, 29898.08, 4610.37, 55120.88, 33261.50, 21418.43;
	31727.72, 41394.83, 35007.70, 48296.68, 2398.01, 24897.11, 56357.20, 46046.97, 1903.76;
	24627.99, 40229.90, 25199.35, 29506.70, 29026.49, 9608.34, 9809.90, 52327.52, 3400.29;
	3519.46, 26957.53, 55348.83, 28387.21, 4591.98, 29809.72, 19091.57, 62287.99, 31026.77;
	37373.47, 7018.38, 57904.46, 51816.14, 29355.54, 14709.94, 19208.40, 50638.96, 31713.48;
	63570.23, 57122.85, 30246.88, 1344.17, 36044.86, 1976.80, 61626.04, 16147.06, 18110.23;
	33778.71, 17326.79, 42631.96, 26421.44, 61714.84, 60056.60, 20435.73, 32796.85, 33003.57;

B = [28149.61; 40733.46; 50621.17; 20058.85; 46682.33; 57341.90; 45629.84; 46817.79; 26234.08]

Starting clock.
Computing Serially.
Stopped clock.

X = [ 0.79; -0.12; -0.62; -0.77; -0.30; -0.20; -0.12;  1.50;  0.96]

Elapsed time = 0.005 ms.
(CPU times are accurate to the nearest 0.001 ms)
My total CPU time for parent = 0 ms.
My system CPU time for parent = 0 ms.
My total CPU time for child processes = 0 ms.
----------------------------------------------
```

Figure 1. Serial code with N=9 and seed =100

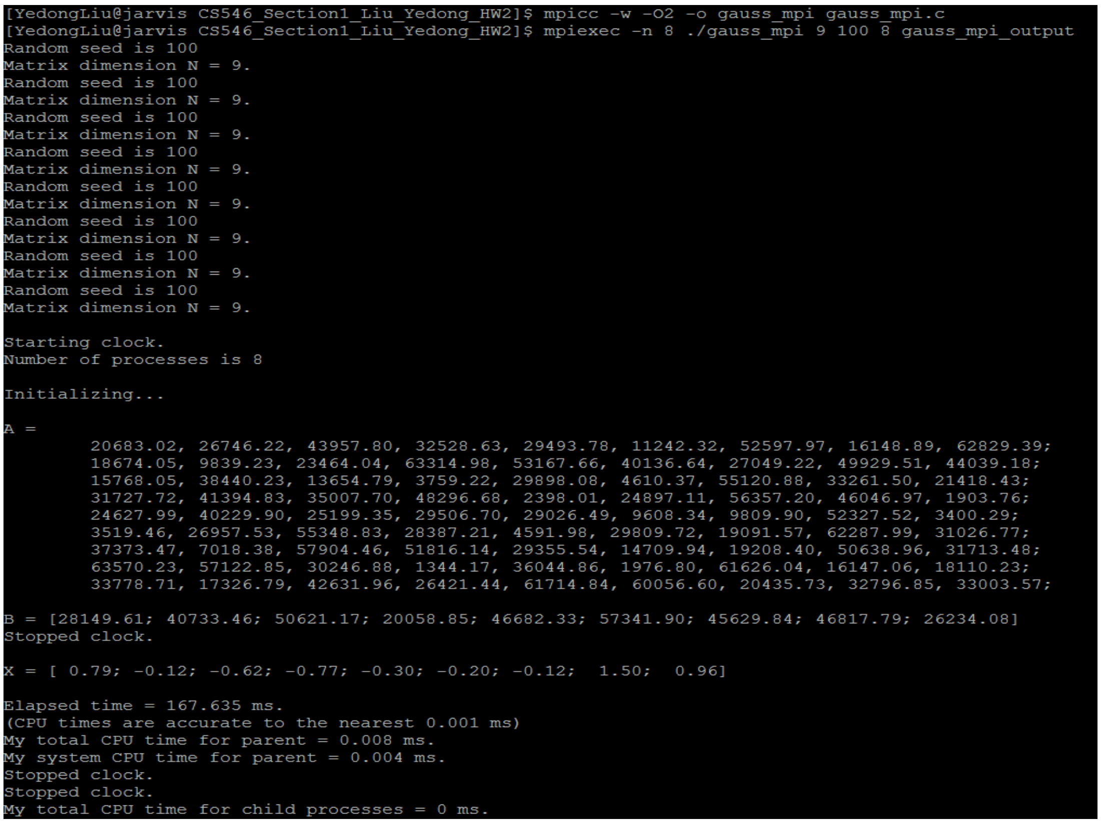
So, for the pthread code, we compile with "gcc -pthread -w -O2 -o gauss_pthread gauss_pthread.c", and we execute with "./gauss_pthread 9 100 8 gauss_pthread_output". The result is the following Figure 2.



```
-------------------------------------------
[YedongLiu@jarvis CS546_Section1_Liu_Yedong_HW2]$ gcc -pthread -w -O2 -o gauss_pthread gauss_pthread.
[YedongLiu@jarvis CS546_Section1_Liu_Yedong_HW2]$ ./gauss_pthread 9 100 8 gauss_pthread_output
Random seed = 100
Computing using 8 threads.
Matrix dimension N = 9.

Initializing...

A =
        20683.02, 26746.22, 43957.80, 32528.63, 29493.78, 11242.32, 52597.97, 16148.89, 62829.39;
        18674.05, 9839.23, 23464.04, 63314.98, 53167.66, 40136.64, 27049.22, 49929.51, 44039.18;
        15768.05, 38440.23, 13654.79, 3759.22, 29898.08, 4610.37, 55120.88, 33261.50, 21418.43;
        31727.72, 41394.83, 35007.70, 48296.68, 2398.01, 24897.11, 56357.20, 46046.97, 1903.76;
        24627.99, 40229.90, 25199.35, 29506.70, 29026.49, 9608.34, 9809.90, 52327.52, 3400.29;
        3519.46, 26957.53, 55348.83, 28387.21, 4591.98, 29809.72, 19091.57, 62287.99, 31026.77;
        37373.47, 7018.38, 57904.46, 51816.14, 29355.54, 14709.94, 19208.40, 50638.96, 31713.48;
        63570.23, 57122.85, 30246.88, 1344.17, 36044.86, 1976.80, 61626.04, 16147.06, 18110.23;
        33778.71, 17326.79, 42631.96, 26421.44, 61714.84, 60056.60, 20435.73, 32796.85, 33003.57;

B = [28149.61; 40733.46; 50621.17; 20058.85; 46682.33; 57341.90; 45629.84; 46817.79; 26234.08]

Starting clock.
Computing Using pthreads with 8 threads.
Stopped clock.

X = [ 0.79; -0.12; -0.62; -0.77; -0.30; -0.20; -0.12;  1.50;  0.96]

Elapsed time = 10.855 ms.
(CPU times are accurate to the nearest 0.001 ms)
My total CPU time for parent = 0.001 ms.
My system CPU time for parent = 0.001 ms.
My total CPU time for child processes = 0 ms.
-------------------------------------------
```

Figure 2. Pthread code with N=9, seed =100 and 8 threads

Similarly, for the MPI program, we compile with ”mpicc -w -O2 -o gauss_mpi gauss_mpi.c”, and execute with “mpiexec -n 8 ./gauss_mpi 9 100 gauss_mpi_output”. The result is shown in the following Figure 3.



```
[YedongLiu@jarvis CS546_Section1_Liu_Yedong_HW2]$ mpicc -w -O2 -o gauss_mpi gauss_mpi.c
[YedongLiu@jarvis CS546_Section1_Liu_Yedong_HW2]$ mpiexec -n 8 ./gauss_mpi 9 100 8 gauss_mpi_output
Random seed is 100
Matrix dimension N = 9.
Random seed is 100
Matrix dimension N = 9.
Random seed is 100
Matrix dimension N = 9.
Random seed is 100
Matrix dimension N = 9.
Random seed is 100
Matrix dimension N = 9.
Random seed is 100
Matrix dimension N = 9.
Random seed is 100
Matrix dimension N = 9.
Random seed is 100
Matrix dimension N = 9.

Starting clock.
Number of processes is 8

Initializing...

A =
        20683.02, 26746.22, 43957.80, 32528.63, 29493.78, 11242.32, 52597.97, 16148.89, 62829.39;
        18674.05, 9839.23, 23464.04, 63314.98, 53167.66, 40136.64, 27049.22, 49929.51, 44039.18;
        15768.05, 38440.23, 13654.79, 3759.22, 29898.08, 4610.37, 55120.88, 33261.50, 21418.43;
        31727.72, 41394.83, 35007.70, 48296.68, 2398.01, 24897.11, 56357.20, 46046.97, 1903.76;
        24627.99, 40229.90, 25199.35, 29506.70, 29026.49, 9608.34, 9809.90, 52327.52, 3400.29;
        3519.46, 26957.53, 55348.83, 28387.21, 4591.98, 29809.72, 19091.57, 62287.99, 31026.77;
        37373.47, 7018.38, 57904.46, 51816.14, 29355.54, 14709.94, 19208.40, 50638.96, 31713.48;
        63570.23, 57122.85, 30246.88, 1344.17, 36044.86, 1976.80, 61626.04, 16147.06, 18110.23;
        33778.71, 17326.79, 42631.96, 26421.44, 61714.84, 60056.60, 20435.73, 32796.85, 33003.57;

B = [28149.61; 40733.46; 50621.17; 20058.85; 46682.33; 57341.90; 45629.84; 46817.79; 26234.08]
Stopped clock.

X = [ 0.79; -0.12; -0.62; -0.77; -0.30; -0.20; -0.12;  1.50;  0.96]

Elapsed time = 167.635 ms.
(CPU times are accurate to the nearest 0.001 ms)
My total CPU time for parent = 0.008 ms.
My system CPU time for parent = 0.004 ms.
Stopped clock.
Stopped clock.
My total CPU time for child processes = 0 ms.
```

Figure 3. MPI code with N=9, seed =100 and 8 processes

As you can see, all the output matrices of X are exactly the same with the original serial code, thus proving the correctness.

Performance

I choose to do calculations from 200 dimensional matrix all the way up to 2000 dimensional matrix with 8 processors or threads for pthread and MPI program. Meanwhile I will do the same calculation for the original serial code to check the speedup.

For Serial code:
Compile with "gcc -O2 -o gauss gauss.c"
Execute with "./gauss <Matrix size> <Random seed> <Output file name>"

For pthread code:
Compile with "gcc -pthread -w -O2 -o gauss_pthread gauss_pthread.c"
Execute with "./gauss_pthread <Matrix size> <Random seed> <Number of threads> <Output file name>"

For MPI code:
Compile with "mpicc -w -O2 -o gauss_mpi gauss_mpi.c"
Execute with "mpiexec -n <Number of processes> ./gauss_mpi <Matrix Size> <Random Seed> <Output file name>"

You can find three original output files in zip, and the record is as shown in Table 1.

Program / Matrix size	Serial (ms)	pthread (ms)	MPI (ms)
200	3.256	65.53	17.588
400	23.097	143.551	40.264
600	73.987	259.437	80.381
800	181.688	430.982	145.155
1000	361.889	694.595	224.936
1200	612.3	809.883	388.989
1400	939.016	1166.17	507.878
1600	1532.38	1370.49	699.784
1800	2227.27	1607.63	932.309
2000	3111.52	1833.53	1238.09

Table 1. Time performance for three codes

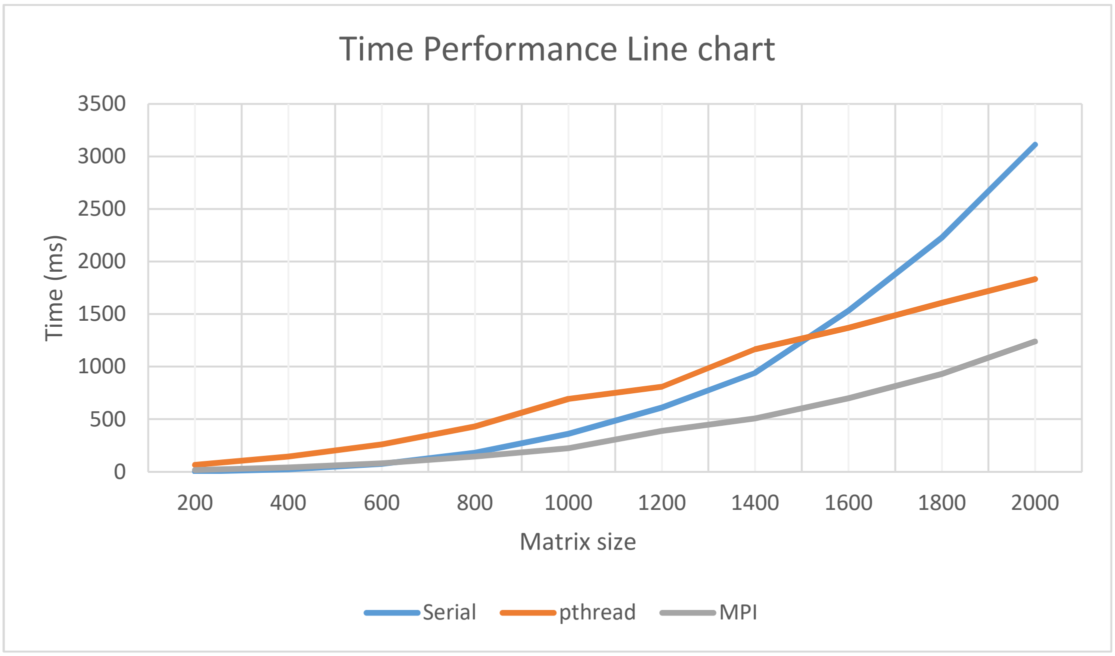


Chart 1. Time performance line chart for three programs

The performance of serial code is pretty good in small dimensional matrix, because at that time, the cost of communications or parallelization is pretty high. After the matrix size increased above 1500, the serial one is slower than both pthread and MPI, and the gap will be bigger and bigger as the matrix size increases. The speedup is 1.69 for pthread and 2.51 for MPI.

Working Process

The original serial code gives us a pretty good example of Gaussian Elimination coding. So I try to start my work based on the original code. There are three nested loops and I work on the inner loops. The pthread code is fine for me, the first version I coded turned out to be working pretty well. But I did meet some challenges for the MPI code. At first, I plan to assign every row of data to every process, after all the processes finished their jobs, each process will send the data back to rank 0. This is the 1.0 version for my MPI code, I tested with N=2000, and the code ran almost 50s to get the result. It is even much slower than the serial code, this is absolutely unacceptable and must be improved. So I changed my mind, rather than sending all data to other processes, this time I only assign specific data to specific process. If this process is in charge of this row, rank 0 will send this row of data to this process, otherwise, it will not send data. So this is version 2.0. This time, the speed is fine but some other issue occur, like there are some deadlock which result in that the program cannot terminate. Moreover, I found another part which can be improved. In the original version, under every normalization line, the row data will be sent back to rank 0 by every other process each time they finished one row calculation. This is unnecessary, the main goal of my improvement is to cut off all the unnecessary communications between processes, so here we only the row[norm+1] has to send its data back to rank 0, all other rows do not need to send back their data because the total elimination has not been finished yet. This is a great idea which reduces my time to around 1500ms when calculating the 2000 dimensional matrix. And I also eliminate the deadlocks and now the program is perfect. The version 3.0 is my final version for MPI.

Future Improvement

There are two parts which I think can be improved.

First one is algorithm. I implemented a pretty simple algorithm to achieve parallelization. I did not implement the pipeline algorithm due to the time limit. So if I had time, I could develop a 2D pipeline version of Gaussian Elimination which I think the performance will be even better. Another idea is that I find that I only parallel in the row loop, inside the row loop is the col loop. This is another place we can do parallel, because the data inside one row calculates independently from each other. If we implement another parallel here, I think the calculation time for the col loop can be reduced. Although I successfully reduced the elapsed time of the whole calculation, the time complexity for my algorithm is still O(n^3), more specifically O($\frac{n^3}{p}$), where p is the number of processes or threads. I did not manage to reduce the time complexity, so I think in the future works, I can try to design a new algorithm which can reduce the time complexity.

The second is the scalability. This is a hard one, because for my program, the scalability is not perfectly shown as it should be. Due to the time limit, I did not find the answer right now but I will try to find out the reason in the future. Here is the scalability table for pthread and MPI with 2000 dimension matrix.

Program / Processes	Pthread (ms)	MPI (ms)
1	4033.9	3062.1
2	3366.85	2115.63
3	2488.81	1422.18
4	1959.9	1252.06
5	1998.63	1309.81
6	2398.7	1339.18
7	2120.7	1350.1
8	2046.82	1359.1

Table 2. Time performance with different threads/processes

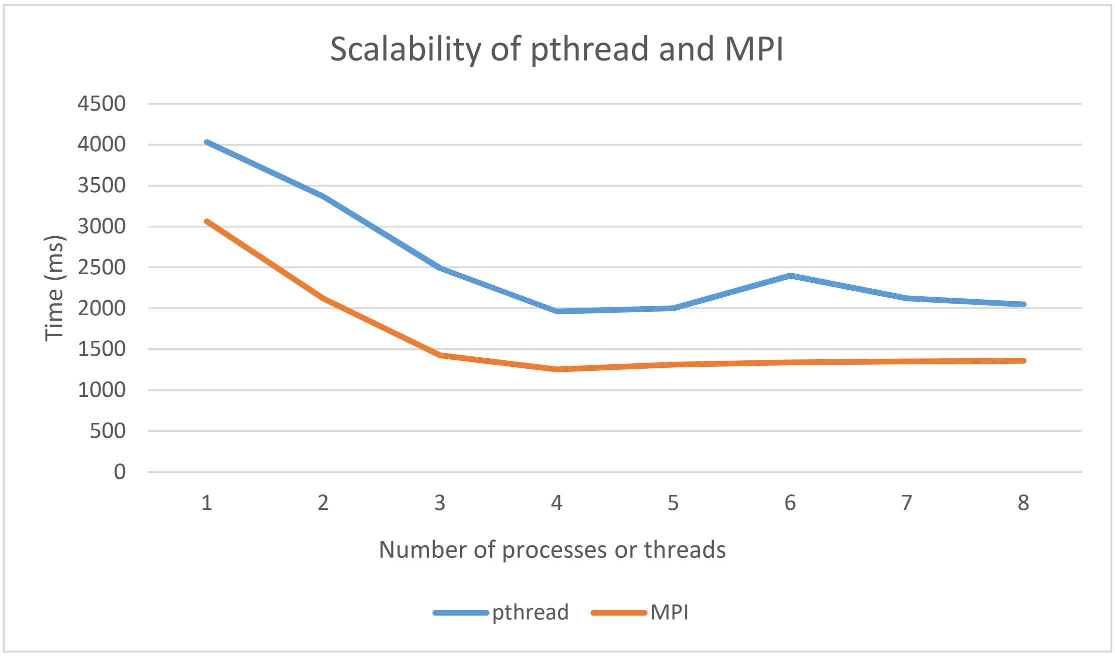


Chart 2. Scalability of pthread and MPI

As you can see, the scalability between 1 and 4 is obvious for both pthread and MPI, but when the number of processes or threads rose above 4, it is not very obvious as it was. So this is another part which I think I can still improve.

Reference

Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar." Introduction to Parallel Computing, Second Edition", ISBN: 0-201-64865-2