The Chinese University of Hong Kong, Shenzhen

CSC4005

Distributed and Parallel Computing

Report for CSC4005 Project 4

Author:

Feng Yutong

Student Number:

120090266

December 5, 2022

Contents

1 Introduction

This program simulates a room with four walls and fire area. The heat will spread from places with higher temperature to places with lower temperature. The project provides sequential, CUDA, MPI, Openmp, pthread and MPI + Openmp versions to simulate the heat spreading process. It uses Jacobi iteration to compute the temperature and using GUI system to show the change in each run. The report compares the performances of six version in the aspect of time, speedup, efficiency under different data size and thread/process number settings.

2 Method

2.1 Sequential version

We maintain two array **data_even** and **data_odd** to avoid data overwritten. Repeat the following step:

Step 1. We calculate the temperate at point (i, j) by

$$H_{i,j} = \frac{H_{i+1,j} + H_{i-1,j} + H_{i,j+1} + H_{i,j-1}}{4}$$

Note that if a point is at the wall, i.e i = 0 / size-1 or j = 0 / size -1, we do not calculate them

Step 2. Maintain fire area: if a point is in fire area, set its temperature to be the fire temperature

Step 3. If all points converge or exceeds maximum iteration, stop the loop.

2.2 MPI version

Broadcast global information such as fire_area, data_odd and data_even to threads. Repeat the following step:

Step 1. Broadcast neighborhood information (one line before the start of sub-job block and one line after sub-job block) from master process (rank = 0) to each process.

Step 2. Each process deals with the same amount of points: update, maintain fire and check continue. The room is divided by the x-axis. The starting x index and ending x index can be calculated by

$$start = \frac{size}{p} \times thread\_id, end = \frac{size}{p} + start$$

where $size$ is number of room width/length and p is the number of processes. The computing process is the same with sequential version.

Step 3: Gather information from each processes to master process, data_even in odd run and data_odd in even run. Gather the OR of all process stopping flag by *MPI_Reduce* to check whether to stop the program. Draw graph in main process.

2.3 Pthread/OpenMP/Cuda version

Step 1. For Cuda version, allocate memory to Cuda. The three versions then use shared memory. Repeat the following steps:

Step 2. Each process deals with the same amount of points: update, maintain fire and check continue. The room is divided by the x-axis. For pthread, the starting x index and ending x index can be calculated by

$$start = \frac{size}{p} \times thread\_id, end = \frac{size}{p} + start$$

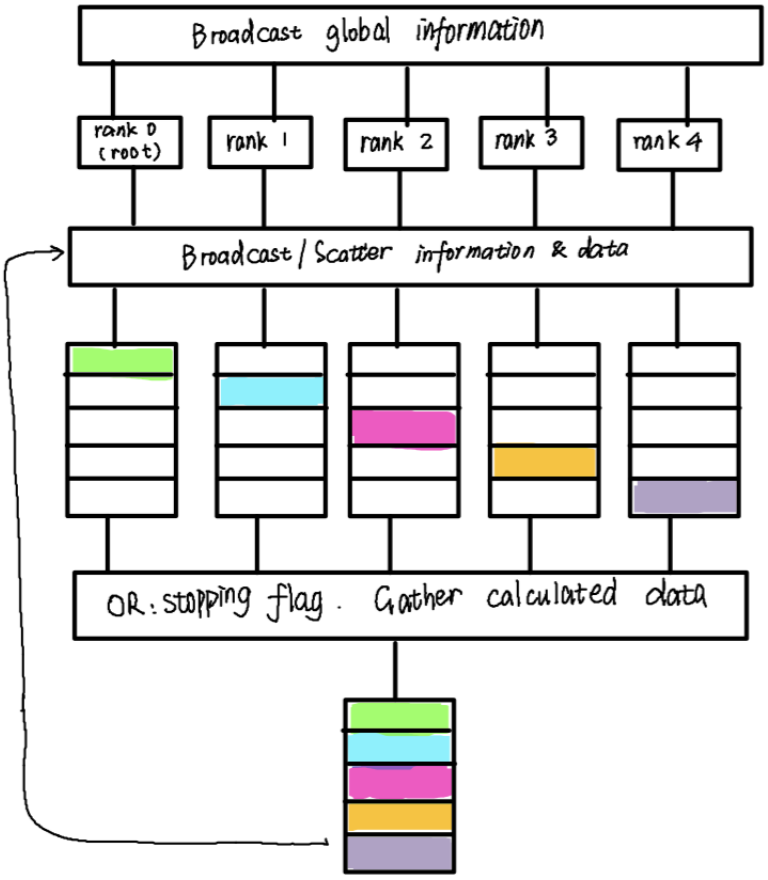

Figure 1: MPI Flow

where *size* is number of room width/length and p is the number of processes. The computing process is the same with sequential version. For Cuda, the index can be calculated by blockDim.x * blockIdx.x + threadIdx.x and it automatically distribute points to different threads. For OpenMP, we use **omp parallel for** to distribute the points. We caculate the OR of all stopping flag to check whether to stop the program.

Step 3. In the main thread we draw the graph. Before that we must block threads to ensure all points in the zoom is updated. For Cuda, we can use **__syncthreads()**; for openmp, we can use **omp barrier**; for pthread, we can use **pthread_join**.

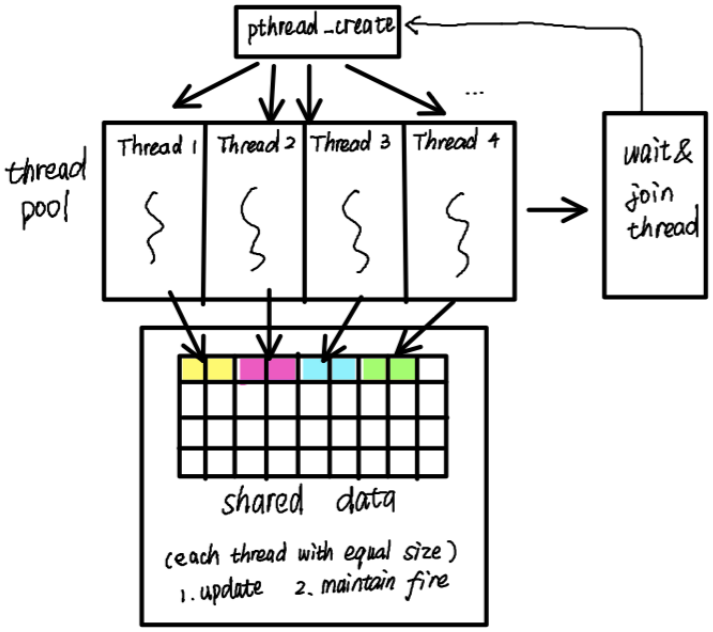

Figure 2: Pthread/OpenMP/Cuda Flow

2.4 MPI+OpenMP version

Broadcast global information such as fire_area, data_odd and data_even to threads. Repeat the following step:

Step 1. Broadcast neighborhood information (one line before the start of sub-job block and one line after sub-job block) from master process (rank = 0) to each process.

Step 2. Each process deals with $\frac{size*size}{p}$ of points as before, where p is the number of processes and size is the width/height of the room. Moreover, divide each process into t threads. For each thread, it deals with $\frac{size*size}{pt}$ of points. Calculation is the same as before.

Step 3: Gather information from each processes to master process, data_even in odd run and data_odd in even run. Gather the OR of all process stopping flag by *MPI_Reduce* to check whether to stop the program. Draw graph in main process.

2.5 Algorithm analysis

Assume the room is of the size: $N \times N$. We analyse the complexity in one run:

1. Sequential version:

The program first computes the temperature of each points, maintain fire area, and then check for the convergence. They all take $O(N^2)$. So the total time complexity is $O(N^2)$.

2. MPI version:

Suppose the number of processes is p. Each sub-process deals with $\frac{N^2}{p}$ points. The program takes $O(N \times 2p)$ to broadcast neighbourhood information and $O(N^2)$ for master process (rank = 0) to gather data to draw the figure. Each process takes

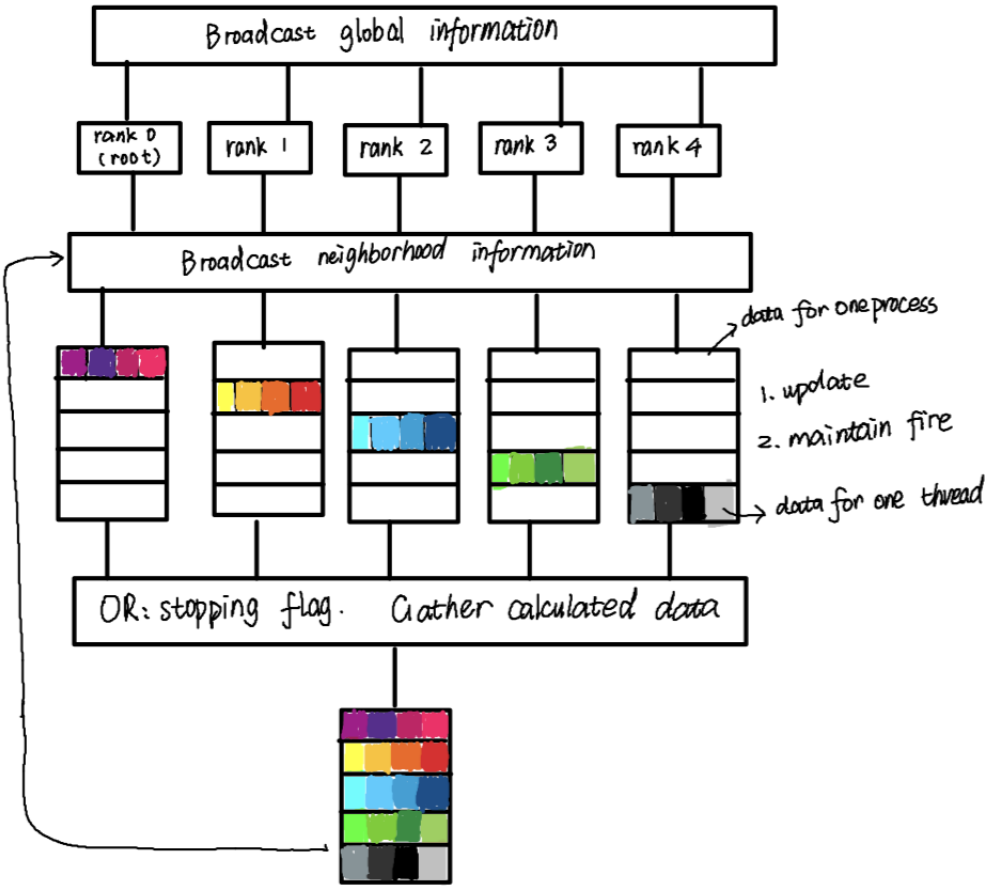

Figure 3: MPI+OpenMP Flow

$O(\frac{N^2}{p})$ to do calculation and stopping flag check. One thing to note is that we only need to broadcast neighbourhood information (size of $O(\frac{2N}{p})$ instead of the whole room information (size of $O(N^2)$). This saves much time. However, we still needs collect all data after each run to draw the figure.

3. Pthread/OpenMP/CUDA version:

Suppose the number of thread is t. The mechanism is similar with MPI, however, they just copied data from the main thread and copied calculated results to main thread. Each thread deals with $\frac{N^2}{t}$ bodies. It takes $O(\frac{N^2}{t})$ to do calculation and stopping flag check and there is no extra communication overhead.

4. MPI+OpenMP version: Suppose the number of thread is t and the number of processes is p. In each run, each thread in its process deals with $\frac{N}{pt}$ points. It takes $O(\frac{N^2}{pt})$ to do calculation and stopping flag check. The program takes $O(N \times 2p)$ to broadcast neighbourhood information and $O(N^2)$ for master process (rank = 0) to gather data to draw the figure.

3 Result

3.1 Execution

Compile by Makefile:

```
make $command
```

where command is one of seq, seqg, mpi, mpig, pthread, pthreadg, cuda, cudag, openmp, openmpg, mpi_openmp, mpi_openmpg.

Run:

Sequential :

```
./seq $problem_size
./seqg $problem_size
```

MPI :

```
mpirun -np $n_processes ./mpi $problem_size
mpirun -np $n_processes ./mpig $problem_size
```

Pthread :

```
./pthread $problem_size $n_threads
./pthreadg $problem_size $n_threads
```

CUDA :

```
./cuda $problem_size
./cudag $problem_size
```

OpenMP :

```
openmp $problem_size $n_omp_threads
openmpg $problem_size $n_omp_threads
```

MPI + OpenMP:

```
mpirun -np $n_processes ./mpi_openmp $problem_size
    $n_omp_threads
mpirun -np $n_processes ./mpi_openmpg $problem_size
    $n_omp_threads
```

Or you can use sbatch to sumbmit the job.

3.2 Result

Here shows five output of the program with 800 * 800 points. The max iteration number is set to 4. The default thread setting for openmp, mpi+openmp, cuda and pthread is 4. For CUDA version, the correctness is checked by comparing the data array after each run with that of sequential version under multiple same large data size setting.

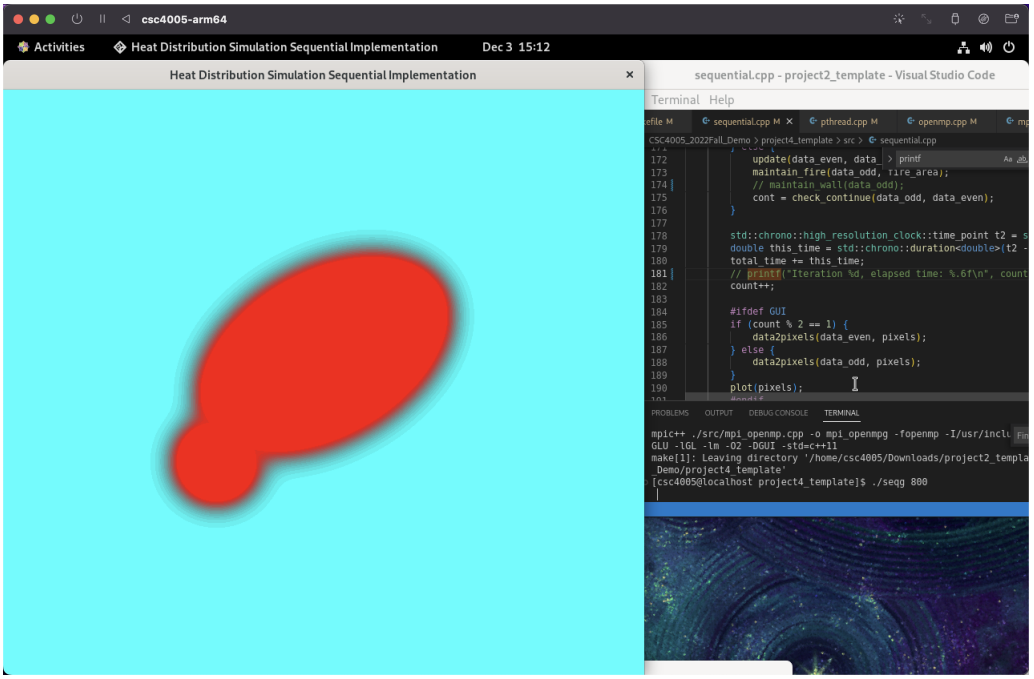

Figure 4: Sequential output

3.3 Performance analysis

3.3.1 Test Design

max_iteration: 5000

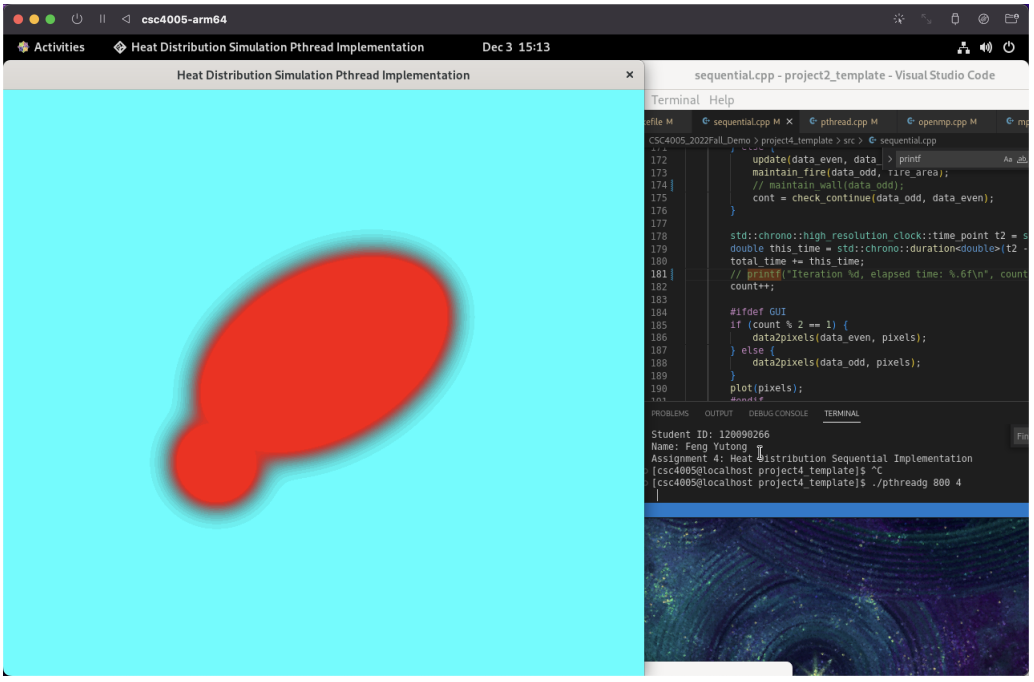

Figure 5: Pthread output

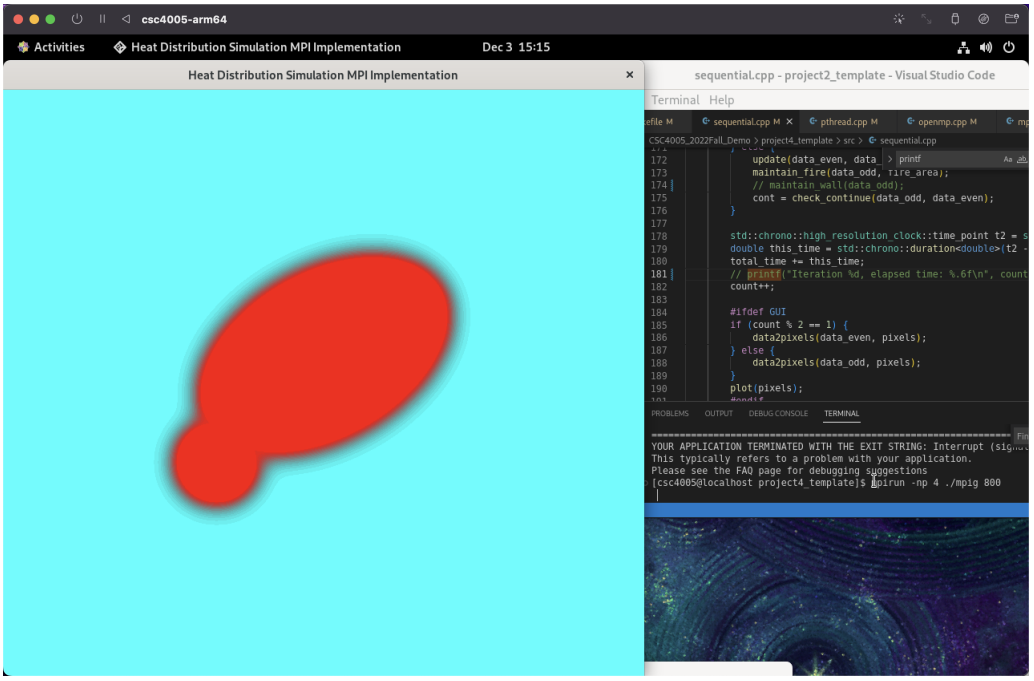

Figure 6: MPI output

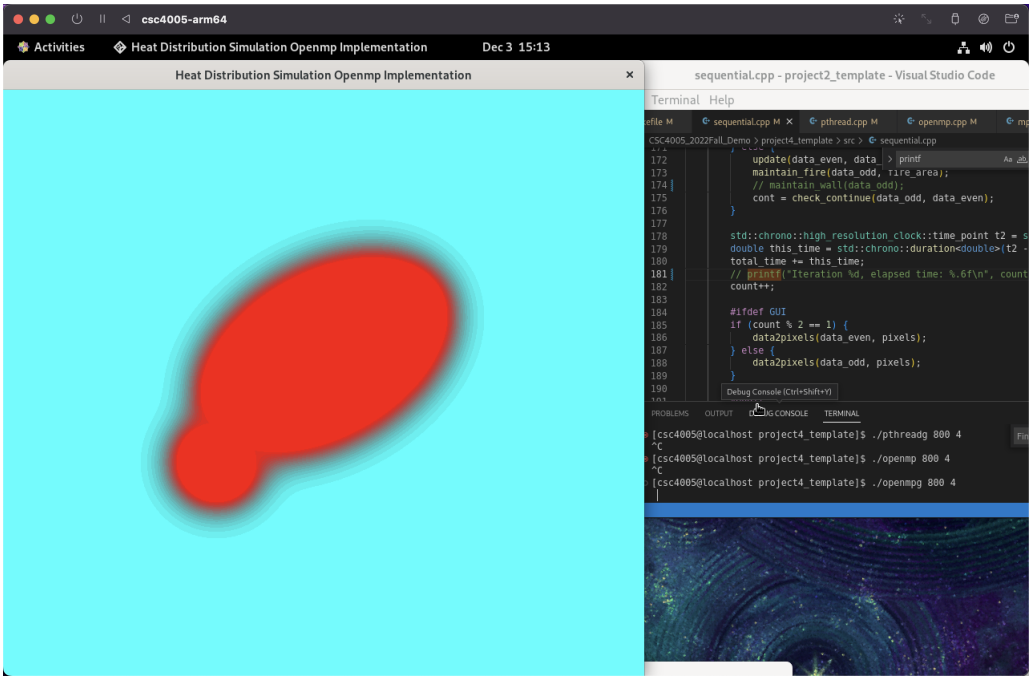

Figure 7: Openmp output

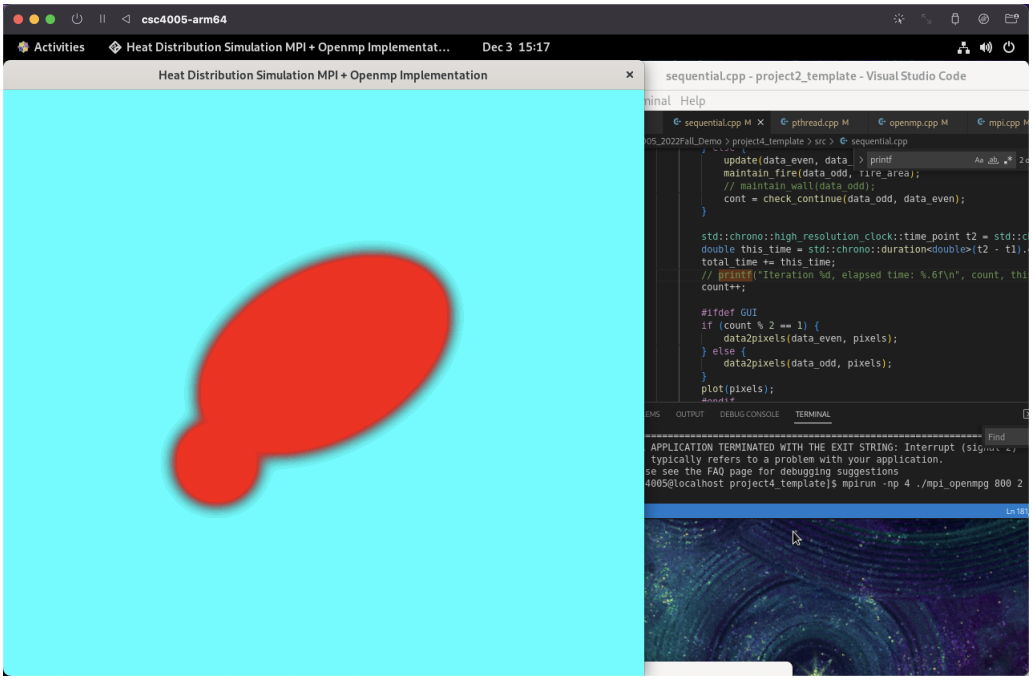

Figure 8: MPI + Openmp output

Data size:200 * 200, 400 * 400, 1000 * 1000, 2000 * 2000

Core/Thread Number: 1, 4, 20, 40

Data is the average of three runs.

3.3.2 MPI VS Pthread VS Sequential VS Cuda VS Openmp VS Openmp + MPI

Figure 10 and Table 1 compares six implementation with different data size (200 * 200, 400 * 400, 1000 * 1000, 2000 * 2000). To compare their performance, thread number and process number is set to 1. From the table we can see that six versions shares similar runtime when data size is small. When data size gets too large, MPI and MPI + Openmp takes much more time which is due to the communication overhead: MPI needs to gather all data information and broadcast neighbour information in addition to normal calculation. CUDA has a better performance in all data sizes since it is designed for high performance calculating.

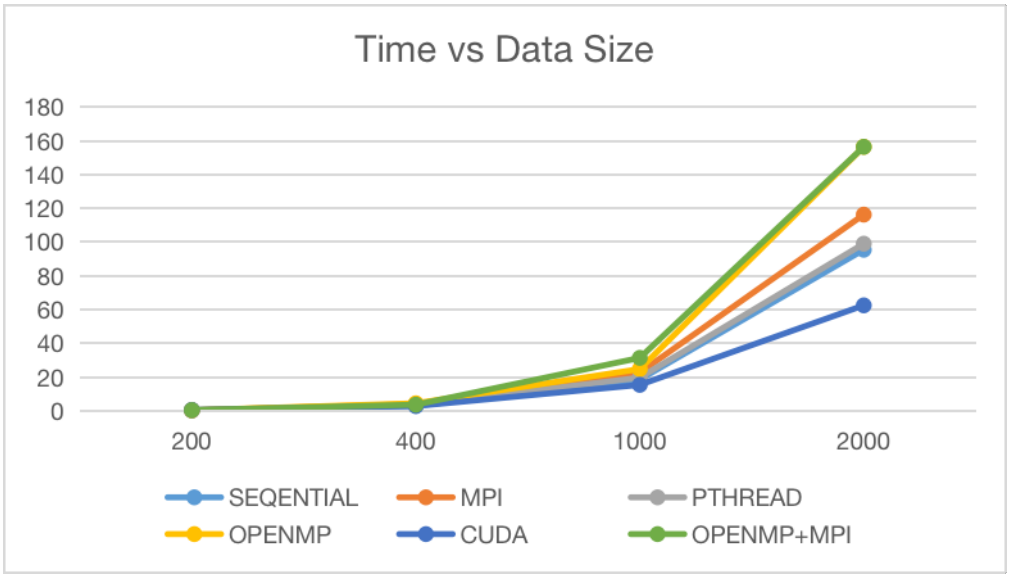

Figure 9: Time VS Thread/Core Number

	200	400	1000	2000
SEQENTIAL	0.191391	2.925111	18.492592	95.453313
MPI	0.246227	3.960581	22.574754	116.313892
PTHREAD	0.304148	3.441455	19.625245	99.064639
OPENMP	0.288706	4.40182	24.703478	156.530769
CUDA	0.469237	2.766611	15.327673	62.490507
OPENMP+MPI	0.365271	3.564564	31.33922	165.342919

Table 1: MPI VS Pthread VS Sequential VS Cuda VS Openmp VS Openmp + MPI

3.3.3 Time

Figure 11 and Table 2 shows Time VS thread/core numbers. The data size is 1000 * 1000. (For MPI + Openmp setting: 2 processes and 2 threads, 5 processes and 4 threads, 8 processes and 5 threads) We can see that the time first decreases with the increasing of thread/core number and then increases with the the increasing of thread/core number. The decrease is due to reduce of calculation time complexity as we analysed in theoretical part. The increase may due to large communication overhead or waiting time to synchronize threads, where its effect on time exceeds calculation. Especially for MPI version, the communication overhead increases greatly when thread number is add up to 40. Though we improve the algorithm by only broadcast neighbour information instead of the whole room's information, we still needs to block threads and collect the whole room's information to draw the graph.

We can further demonstrate the effect by comparing speedup and efficiency.

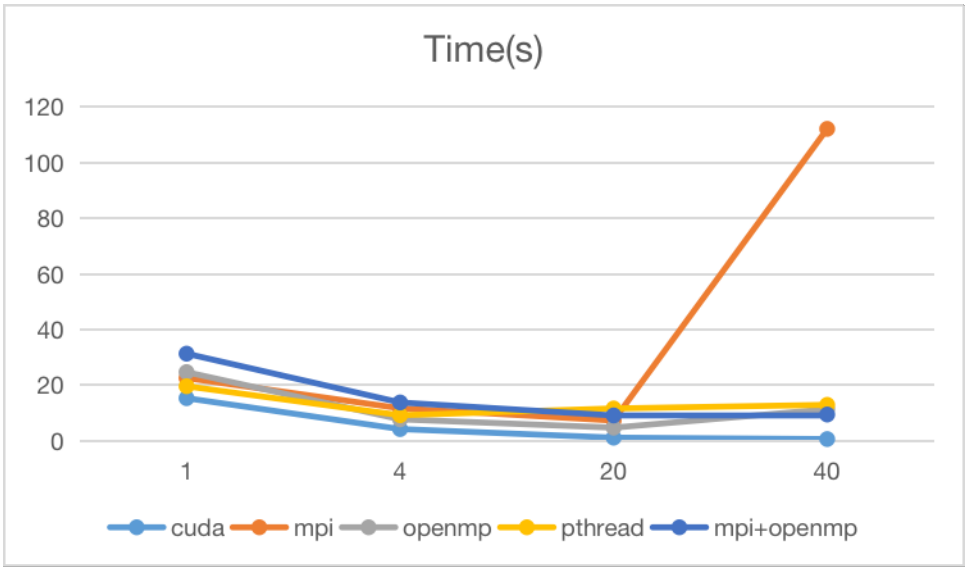

Figure 10: Time VS Thread/Core Number

Thread Number	CUDA	MPI	OPENMP	PTHREAD	MPI_OPENMP
1	15.327673	22.574754	24.703478	19.625245	31.33922
4	4.22032	11.704075	7.724513	9.199585	13.781975
20	1.127113	7.288312	4.724487	11.628534	9.113214
40	0.628615	112.163667	11.341275	12.92895	9.447399

Table 2: Time based on different thread/core Number

3.3.4 Speedup

The speedup is given as:

$$Speedup = \frac{\text{running time on one server}}{\text{running time on parallel server}}$$

Figure 12 and Table 3 shows Speedup VS thread/core numbers. We can see that except for CUDA version, the speedup first increases with the increasing of thread-/core number and then decreases with the the increasing of thread/core number. That is because smaller sub-task size takes less runtime and when the thread/process number is large enough, ratio of communication time and waiting time over calculation time becomes larger.

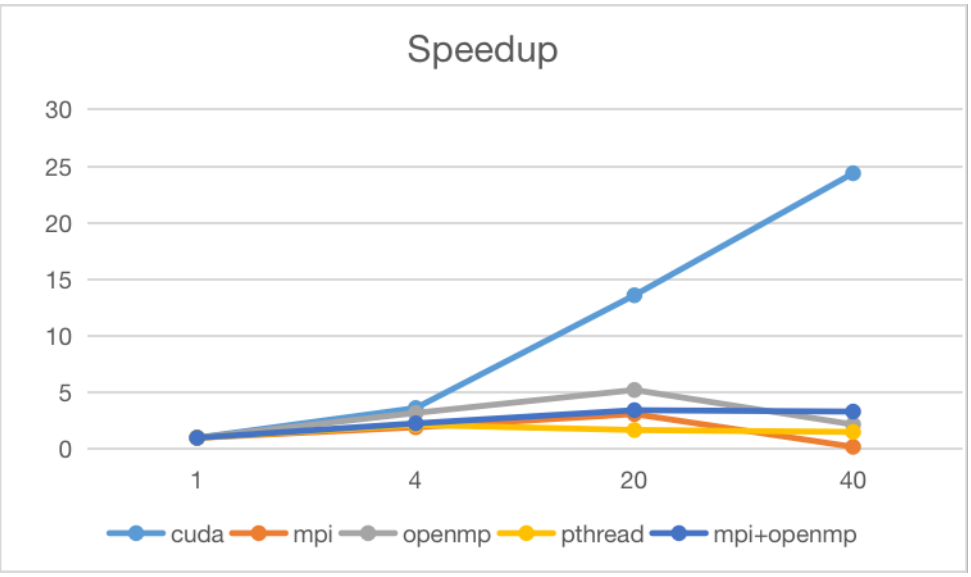

Figure 11: Speedup based on different thread/core number

3.3.5 Efficiency

Efficiency is given as

$$Efficiency = \frac{\text{Speedup}}{\text{process number}}$$

Thread Number	CUDA	MPI	OPENMP	PTHREAD	MPI_OPENMP
1	1	1	1	1	1
4	3.631874597	1.928794373	3.198062842	2.133275034	2.273928084
20	13.59905617	3.097391275	5.228817012	1.687680064	3.438876778
40	24.38324412	0.201266191	2.178192311	1.517930304	3.317232606

Table 3: Speedup based on different thread/core number

Figure 13 and Table 4 shows Efficiency VS thread/core numbers. Efficiency drops with the increasing of process/thread number for all data size. One possible reason is that when process/thread number is larger, the data size distributed to sub-process decreases, so the real computing time proportion drops. Meanwhile, more time is used for communication overhead and synchronization, e.g.

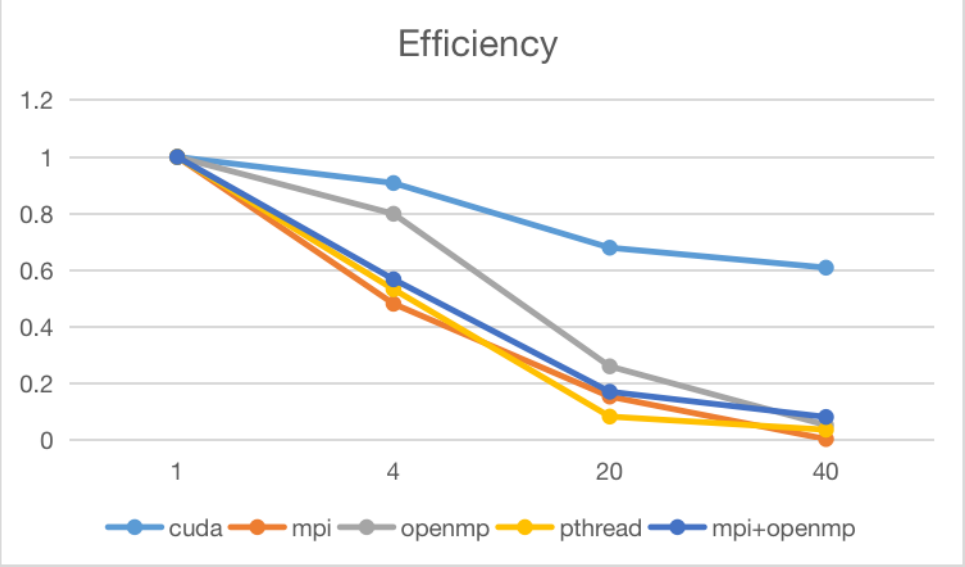

Figure 12: Efficiency based on different thread/core number

Thread Number	CUDA	MPI	OPENMP	PTHREAD	MPI_OPENMP
1	1	1	1	1	1
4	0.907968649	0.482198593	0.799515711	0.533318758	0.568482021
20	0.679952809	0.154869564	0.261440851	0.084384003	0.171943839
40	0.609581103	0.005031655	0.054454808	0.037948258	0.082930815

Table 4: Efficiency based on different thread/core number

3.3.6 Bonus: Openmp VS MPI VS Openmp

Since Cuda/Openmp/Pthread are multi-threads, MPI is multi-processes, we can combine MPI and Openmp together. This part compare multi-processes (MPI), multi-thread (Openmp) and their combination (MPI + Openmp). Figure 14-15 and Table 5-7 show Time VS Data size based on different process/thread number. For Openmp + MPI, we have two sets: one with 4 processes, the other with 10 processes. We can see that 20 threads/processes have better performance in three versions. Less threads performs worse for more computing time and More threads performs worse for communication overhead or synchronization waiting time. For MPI, too much processes (40) is even worse than single process due to the communication overhead. Openmp uses shared memory, so it performs better than MPI in general. Again, we can fix data size to 1000 * 1000 (see Figure 17 and Table 8), we see MPI + Openmp and openmp shares similar performance and MPI in 40 processes has a large increase in time. This is because openmp uses share memory to reduce the effect of communication for a fixed process(thread) number.

4 Conclusion

This report discusses three parallel modes, multi-thread (pthread, CUDA, Openmp), multi-processes (MPI), combination of multi-thread and multi-processes (MPI +

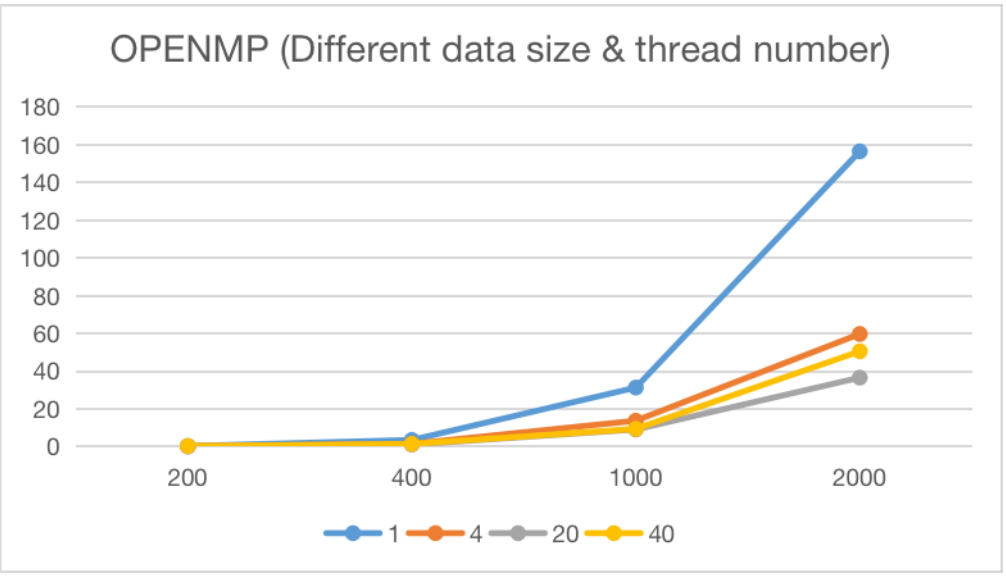

Figure 13: Time vs Thread Number – Openmp

Process Number-Data Size	200	400	1000	2000
1	0.288706	4.40182	24.703478	156.530769
4	0.108086	0.813569	7.724513	36.048029
20	1.506707	2.244128	4.724487	14.120446
40	1.927432	4.309529	11.341275	24.405796

Table 5: Time vs Thread Number – Openmp

Process Number-Data Size	200	400	1000	2000
1	0.246227	3.960581	22.574754	116.313892
4	0.204593	2.381016	11.704075	57.362545
20	0.297139	1.246142	7.288312	29.501884
40	5.227975	24.879505	112.163667	304.241095

Table 6: Time vs Process Number – MPI

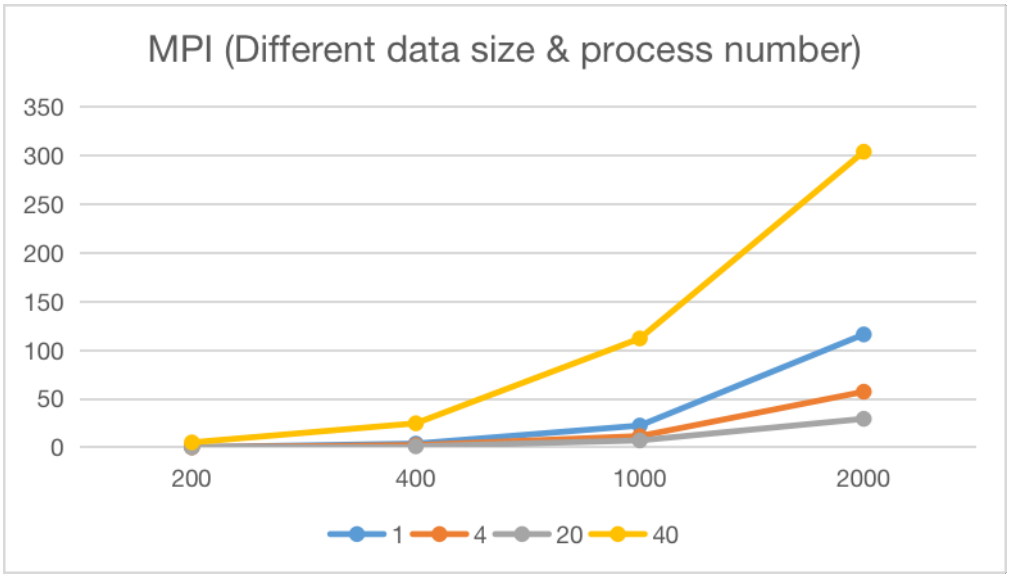

Figure 14: Time vs Process Number – MPI

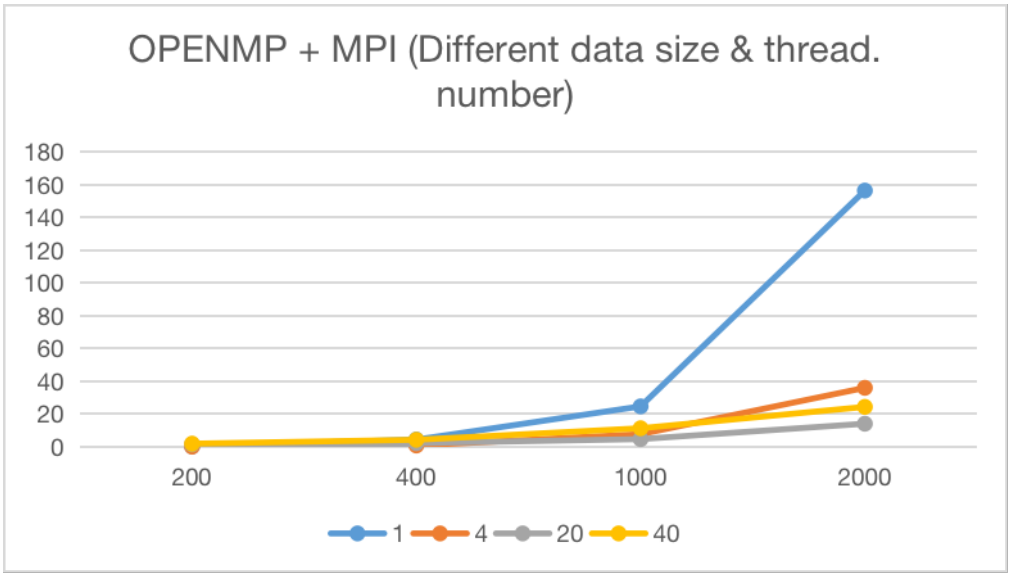

Figure 15: Time vs Thread Number – Openmp + MPI

Process Number-Data Size	200	400	1000	2000
1	0.365271	3.564564	31.33922	165.342919
4	0.231683	1.528414	13.781975	59.743049
20	0.167171	1.15124	9.113214	36.624381
40	0.340614	1.475085	9.447399	50.561015

Table 7: Time vs Thread Number – Openmp + MPI (4)

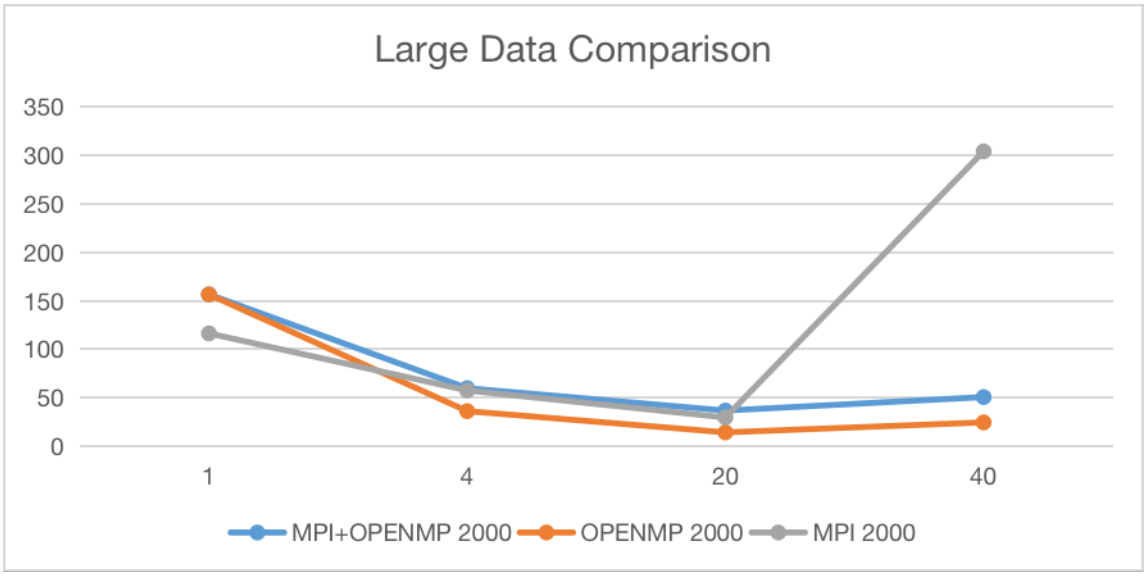

Figure 16: Time Comparison based on Large Data

Process Number	MPI+OPENMP 2000	OPENMP 2000	MPI 2000	2000
1	156.530769	156.530769	116.313892	116.313892
4	59.743049	36.048029	57.362545	57.362545
20	36.624381	14.120446	29.501884	29.501884
40	50.561015	24.405796	304.241095	304.241095

Table 8: Time Comparison based on Large Data

openmp). We compare their performance based on different data size, process/thread number by analysing time, speedup and efficiency. Multi-thread program uses shared memory, while multi-process program takes extra time to do communication. Thus, multi-thread programs perform better in general. Besides, parallel mode can accelerate the runtime under large data size, however, the trade-off between computation and communication as well as synchronization should be consider when choosing thread/process number.