The Chinese University of Hong Kong, Shenzhen

CSC4005

Distributed and Parallel Computing

Report for CSC4005 Project 3

Author:

Feng Yutong

Student Number:

120090266

November 16, 2022

Contents

1 Introduction

2 Method

2.1 Sequential version

Step 1. We compute the forces between two different bodies, which is given by

$$F = \frac{Gm_i m_j}{r^2}$$

Then we sum up the forces and calculated the acceleration, which is given by

$$a = \sum \frac{F}{m_i}$$

Note that it two bodies collide with each other, we ignore the forces. Then we update the velocity of body i:

$$v = v + \Delta t a$$

Step 2. Update the position, which is given by

$$x = x + \Delta t v$$

If two bodies collide, reverse the directions of their velocities. If a body hits the wall, reverse the velocity and adjust its position within the window.

Step 3. After updating position, draw the graph.

2.2 MPI version

Step 1. Broadcast information from master process (rank = 0) to each process, including m, x, y, vx, vy.

Step 2. Each process deals with the same amount of bodies. The starting index and ending index can be calculated by

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

where n is number of bodies 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, including x, y, vx, vy. Draw graph in main process.

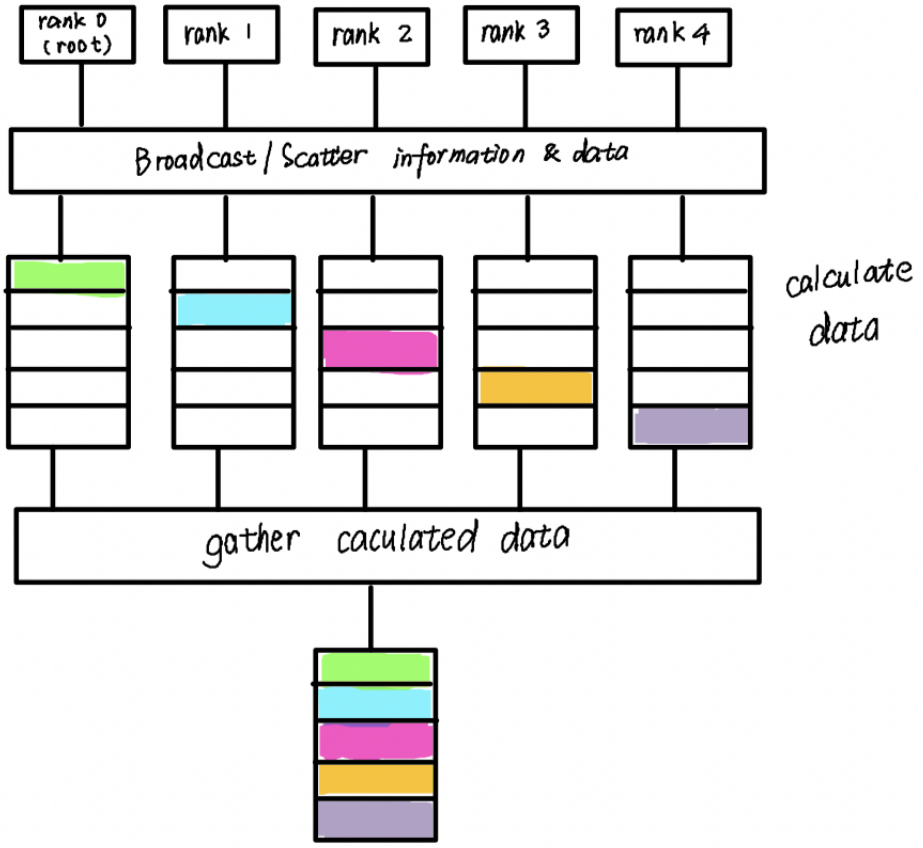

Figure 1: MPI Flow

2.3 Pthread/OpenMP/Cuda version

Step 1. For Cuda version, allocate memory and copy data to Cuda. The three then use shared memory.

Step 2. Each thread deals with the same amount of bodies. For pthread, the starting index and ending index can be calculated by

$$start = \frac{n}{p} \times block, end = \frac{n}{p} + start$$

where n is number of bodies and t is the number of threads. 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 bodies to different threads. For OpenMP, we use **omp parallel for** to distribute the bodies. Since we uses position to calculate the velocity, we must block threads before updating posistions. For Cuda, we can use **__syncthreads()**; for openmp, we can use **omp barrier**; for pthread, we can use **pthread_barrier_wait**.

Step 3. After updating position, draw the graph.

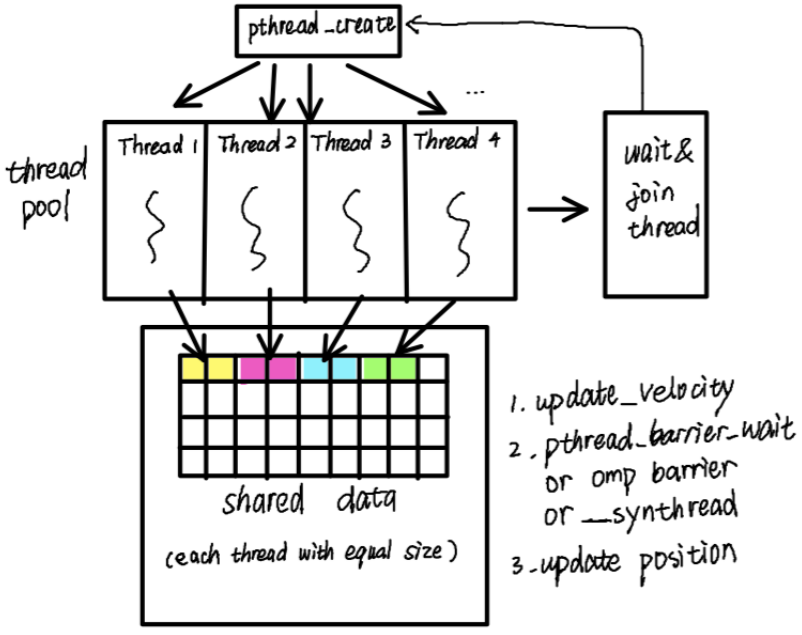

Figure 2: Pthread/OpenMP/Cuda Flow

2.4 MPI+OpenMP version

Step 1. Broadcast information from master process (rank = 0) to each process, including m, x, y, vx, vy.

Step 2. Divide each process into t threads. For each thread, it deals with $\frac{n}{pt}$ bodies, where n is number of bodies and p is the number of processes. Calculation is the same as before.

Step 3. Gather information from each processes to master process, including x, y, vx, vy. Draw graph in main process. Threads in each process use shared memory so there is no extra communication overhead.

2.5 Algorithm analysis

Assume there are N bodies.

1. Sequential version:

The program first computes forces between two vertices, which takes $O(N^2)$. Then it updates the velocity, and and location of all bodies. Here we consider the collisions with other bodies $O(N^2)$ and with the wall $O(N)$. So total time complexity is $O(N^2)$

2. MPI version:

Suppose the number of processes is p. Each sub-process deals with $\frac{N}{p}$ bodies. The program takes $O(N)$ for master process (rank = 0) to broadcast and gather information form other slave processes. Each process takes $O(\frac{N^2}{p})$ to calculate the forces between two bodies. Here we consider the collisions with other bodies $O(\frac{N^2}{p})$ and with the wall $O(\frac{N}{p})$. So total time complexity is $O(\frac{N^2}{p})$.

3. Pthread/OpenMP/CUDA version:

Suppose the number of thread is t. The mechanism is similar with MPI, however,

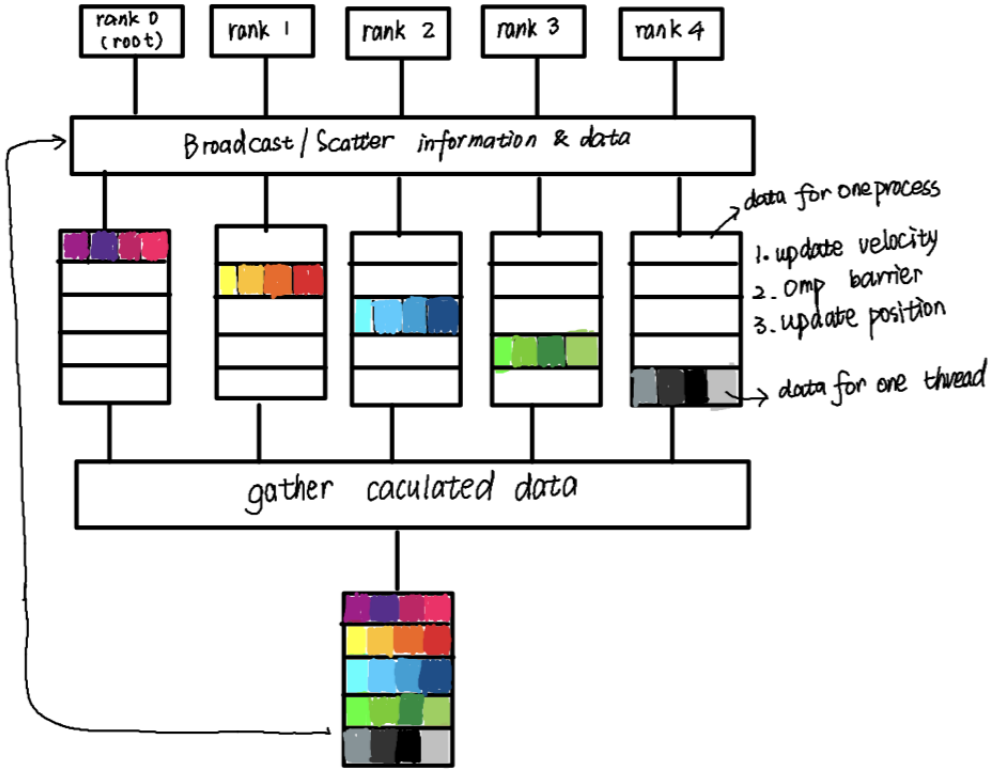

Figure 3: MPI+OpenMP Flow

they just copied data from the main thread and copied calculated results to main thread. The communication overhead is $O(N)$. Each thread deals with $\frac{N}{t}$ bodies. It takes $O(\frac{N^2}{t})$ to calculate the forces between two bodies. Here we consider the collisions with other bodies $O(\frac{N^2}{t})$ and with the wall $O(\frac{N}{t})$. So total time complexity is $O(\frac{N^2}{t})$.

4. MPI+OpenMP version: Suppose the number of thread is t and the number of processes is p. First MPI takes $O(N)$ for master process (rank = 0) to broadcast and gather information form other slave processes. Then each thread in its process deals with $\frac{N}{pt}$ bodies. It takes $O(\frac{N^2}{t})$ to calculate the forces between two bodies. Here we consider the collisions with other bodies $O(\frac{N^2}{pt})$ and with the wall $O(\frac{N}{pt})$. So total time complexity is $O(\frac{N^2}{pt})$.

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 $n_body $n_iterations
./seqg $n_body $n_iterations
```

MPI :

```
mpirun -np $n_processes ./mpi $n_body $n_iterations
mpirun -np $n_processes ./mpig $n_body $n_iterations
```

Pthread :

```
./pthread $n_body $n_iterations $n_threads
./pthreadg $n_body $n_iterations $n_threads
```

CUDA :

```
./cuda $n_body $n_iterations
./cudag $n_body $n_iterations
```

OpenMP :

```
openmp $n_body $n_iterations $n_omp_threads
openmpg $n_body $n_iterations $n_omp_threads
```

MPI + OpenMP:

```
mpirun -np $n_processes ./mpi_openmp $n_body
    $n_iterations $n_omp_threads
mpirun -np $n_processes ./mpi_openmpg $n_body
    $n_iterations $n_omp_threads
```

Or you can use sbatch to sumbmit the job.

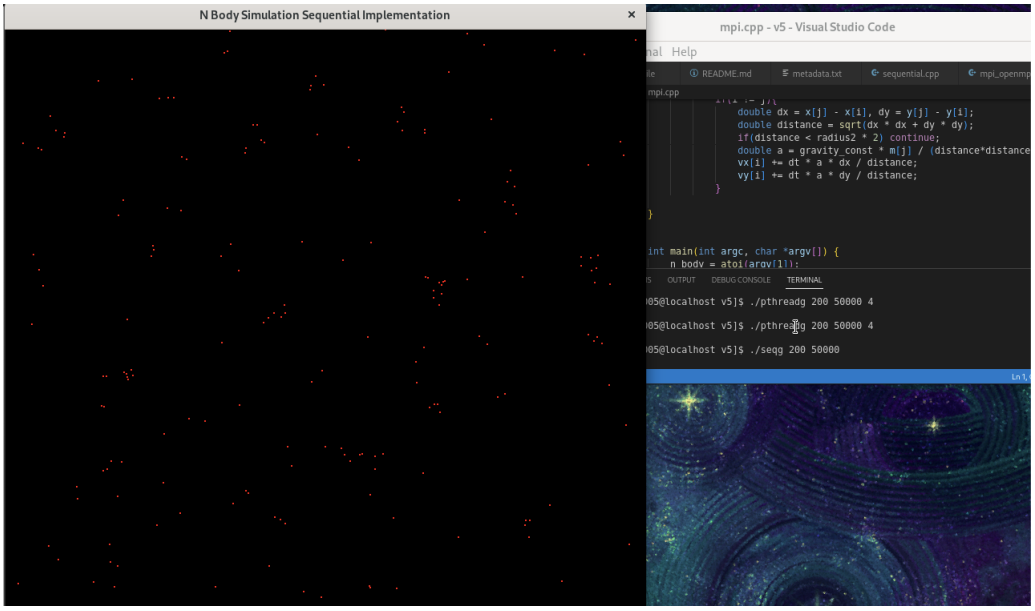

Figure 4: Sequential output

3.2 Result

Here shows six output of the program with 200 bodies, 800 * 800 size image. The default thread setting for openmp, mpi+openmp, cuda and pthread is 4.

3.3 Performance analysis

3.3.1 Test Design

max_iteration: 100

Data size:200, 1000, 5000, 10000

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

Data is the average of three runs.

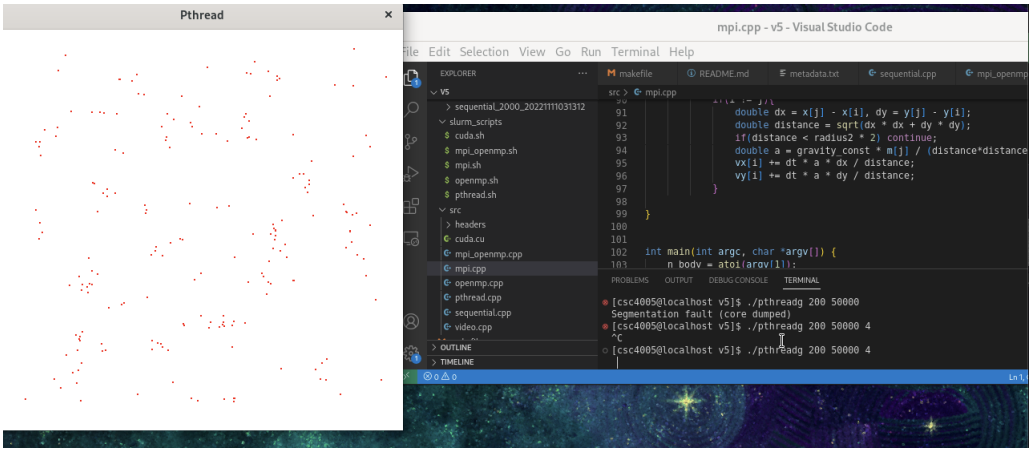

Figure 5: Pthread output

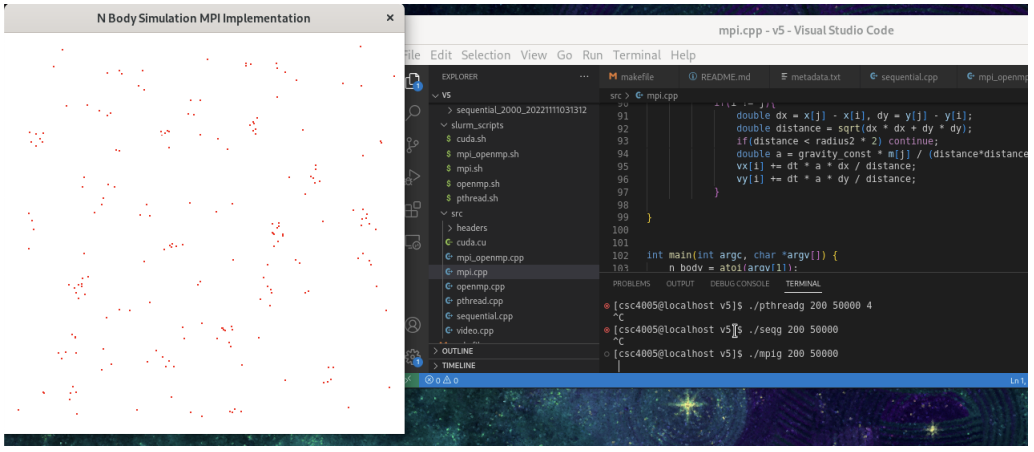

Figure 6: MPI output

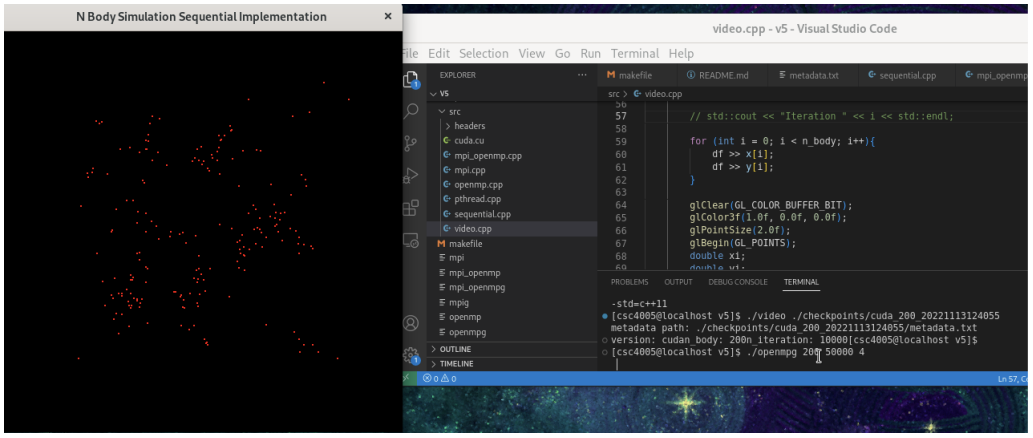

Figure 7: Openmp output

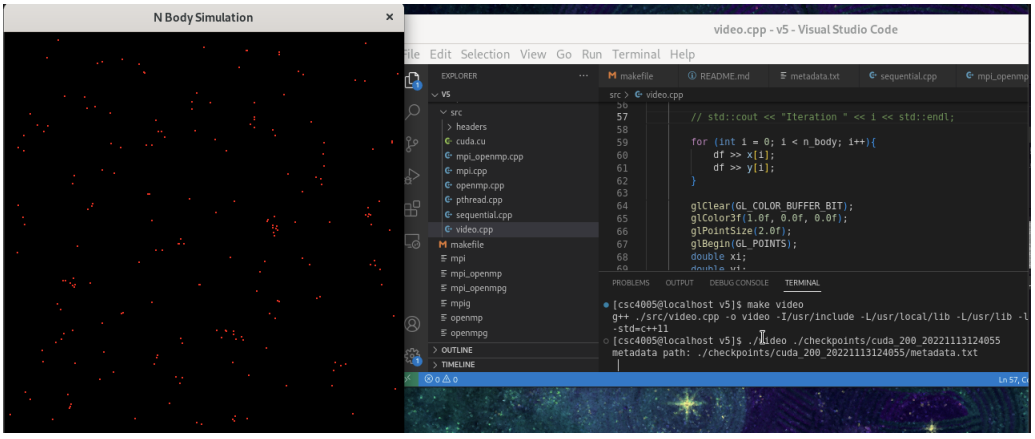

Figure 8: Cuda output

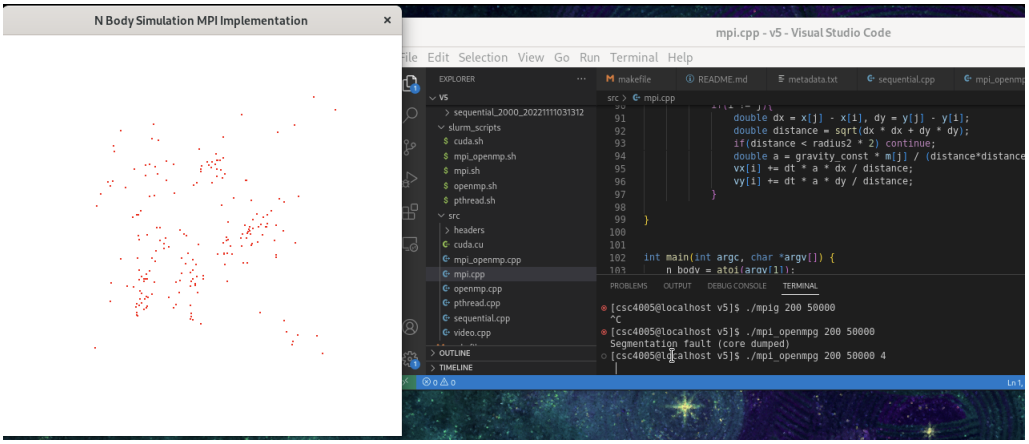

Figure 9: MPI + Openmp output

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, 1000, 5000, 10000). To compare their performance, thread number and process number is set to 1. (MPI + Openmp has default 4 threads; Cuda cannot set to have a thread containing block size with 10000) From the table we can see that sequential version is the fastest under different datasets, which is because sequential does not need to copy memory or communicate between threads/processes. Others have similar time.

SEQENTIAL	0.056	1.172	27.434	112.91
MPI	0.057	1.194	28.814	113.94
PTHREAD	0.092	1.356	30.366	120.582
OPENMP	0.058	1.226	30.05	117.722
CUDA	0.427	1.26	30.447	
OPENMP+MPI	0.056	1.226	30.712	116.657

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

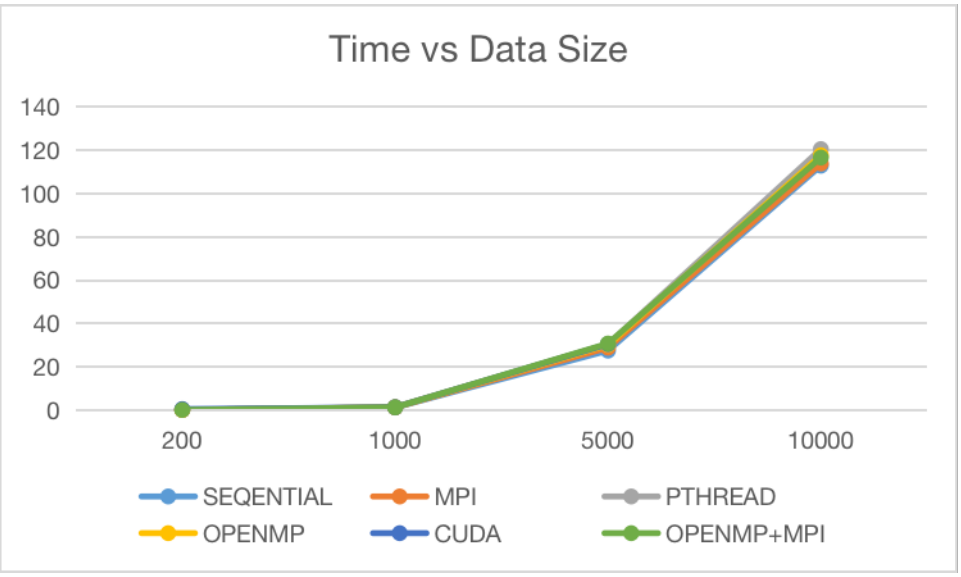

Figure 10: MPI VS Pthread VS Sequential VS Cuda VS Openmp VS Openmp + MPI

3.3.3 Time

Figure 11 and Table 2shows Time VS thread/core numbers. The data size is 1000. (MPI + Openmp has default 4 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 time complexity as we analysed in theoretical part. The increase is due to large communication overhead, where its effect on time exceeds communication.

Thread Number	CUDA	MPI	OPENMP	PTHREAD	MPI_OPENMP
1	3.946	1.194	1.226	1.356	0.534
4	1.276	0.34	0.524	0.482	0.157
20	0.566	0.113	0.369	0.319	2.872
40	0.573	1.514	2.441	0.396	10.18

Table 2: MPI VS Pthread VS Sequential VS Cuda VS Openmp VS Openmp + MPI

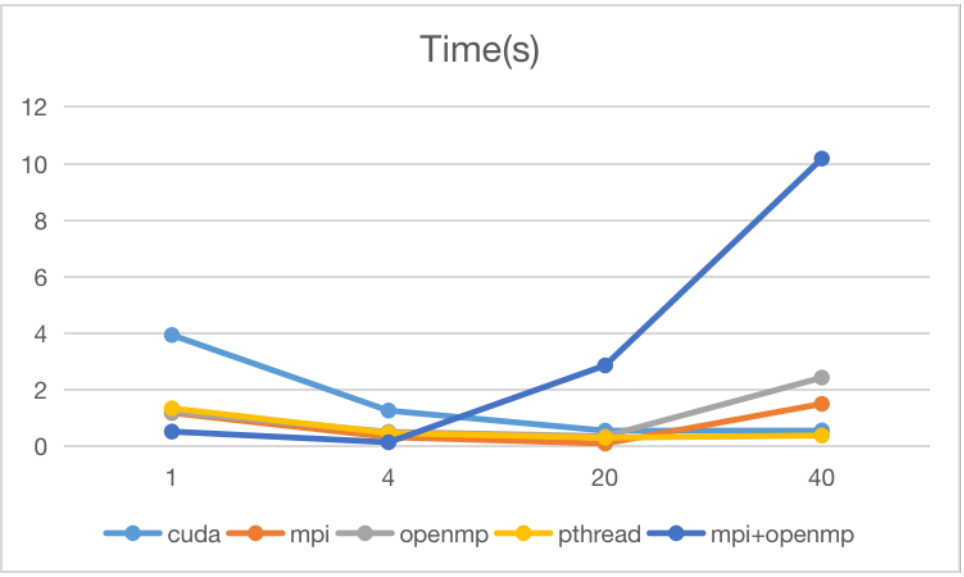

Figure 11: Time VS Thread/Core Number

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

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 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 over calculation time becomes larger.

3.3.5 Efficiency

Efficiency is given as

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

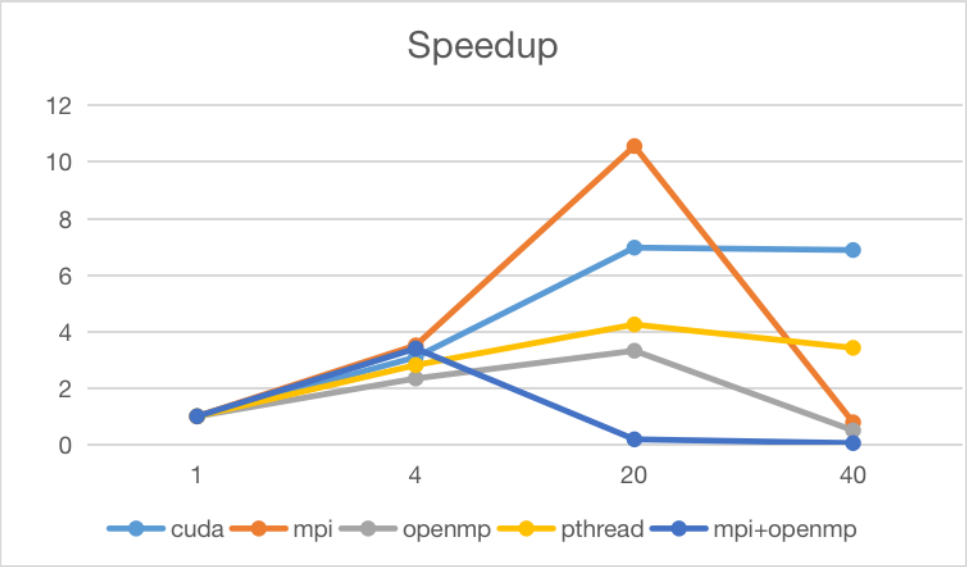

Figure 12: Speedup based on different thread/core number

Thread Number	CUDA	MPI	OPENMP	PTHREAD	MPI_OPENMP
1	1	1	1	1	1
4	3.092476489	3.511764706	2.339694656	2.813278008	3.401273885
20	6.971731449	10.56637168	3.322493225	4.250783699	0.185933148
40	6.886561955	0.788639366	0.502253175	3.424242424	0.052455796

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, e.g.

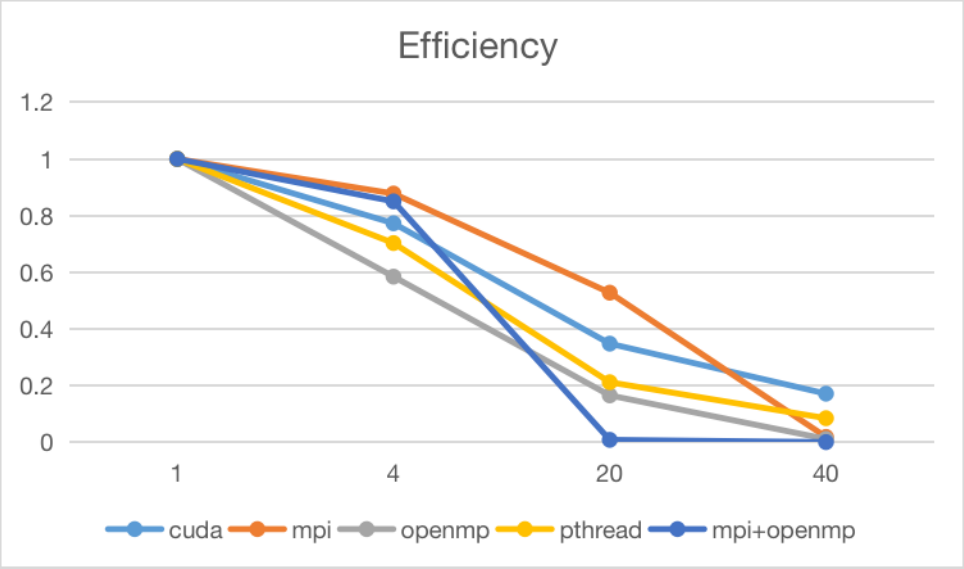

Figure 13: Efficiency based on different thread/core number

Thread Number	CUDA	MPI	OPENMP	PTHREAD	MPI_OPENMP
1	1	1	1	1	1
4	0.773119122	0.877941176	0.584923664	0.703319502	0.850318471
20	0.348586572	0.528318584	0.166124661	0.212539185	0.009296657
40	0.172164049	0.019715984	0.012556329	0.085606061	0.001311395

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 14-17 and Table 5-8 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 4 threads have better performance in Openmp, and Openmp + MPI(4) while 20 threads have better performance in MPI, Openmp + MPI(10). Less threads performs worse for more computing time and More threads performs worse for large communication overhead. Again, we can fix data size to 10000 (see Figure 18 and Table 9), we see the time performance is similar to what we analysed before: there is a trade-off between communication and computation.

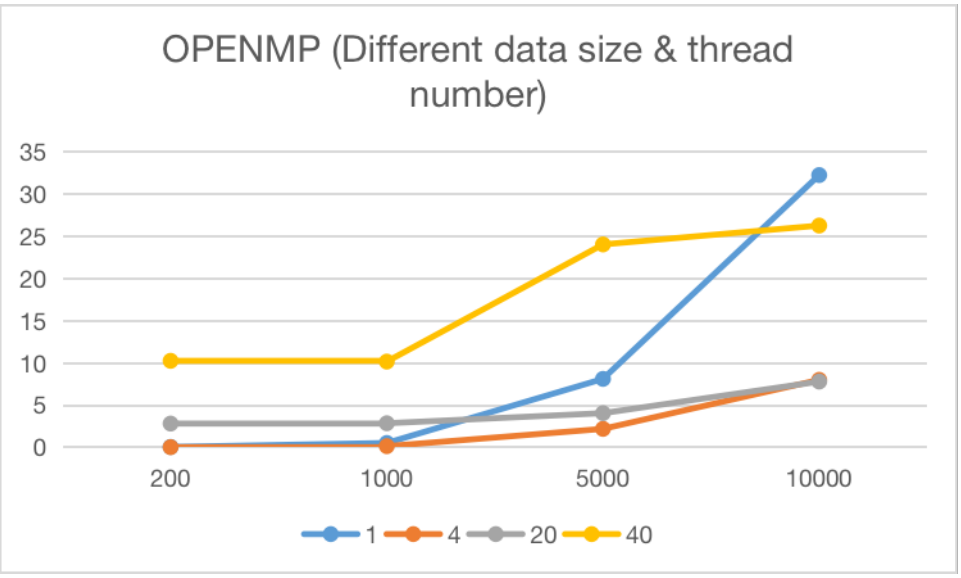

Figure 14: Time vs Thread Number – Openmp

Process Number-Data Size	200	1000	5000	10000
1	0.058	1.226	30.05	117.722
4	0.07	0.524	7.858	30.436
20	0.289	0.369	2.55	9.107
40	2.224	2.441	4.291	8.205

Table 5: Time vs Thread Number – Openmp

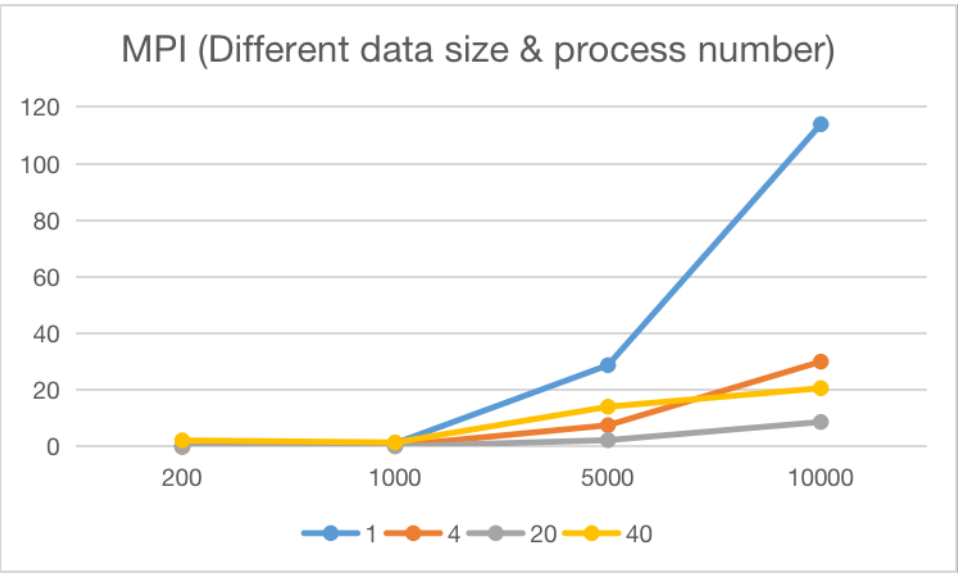

Figure 15: Time vs Process Number – MPI

Process Number-Data Size	200	1000	5000	10000
1	0.05	1.194	28.814	113.94
4	0.025	0.34	7.583	30.059
20	0.014	0.113	2.322	8.737
40	2.241	1.514	14.133	20.677

Table 6: Time vs Process Number – MPI

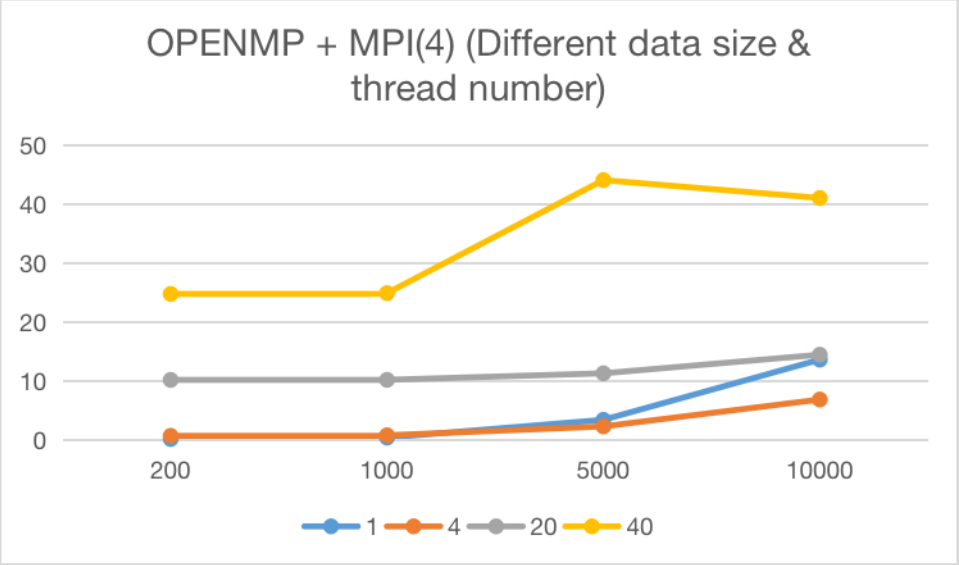

Figure 16: Time vs Thread Number – Openmp + MPI (4)

Process Number-Data Size	200	1000	5000	10000
1	0.065	0.534	8.131	32.285
4	0.04	0.157	2.208	8.017
20	2.834	2.872	4.067	7.813
40	10.271	10.18	24.066	26.302

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

Process Number-Data Size	200	1000	5000	10000
1	0.153	0.39	3.42	13.65
4	0.692	0.786	2.295	6.868
20	10.207	10.216	11.327	14.491
40	24.823	24.962	44.185	41.143

Table 8: Time vs Thread Number – Openmp + MPI (10)

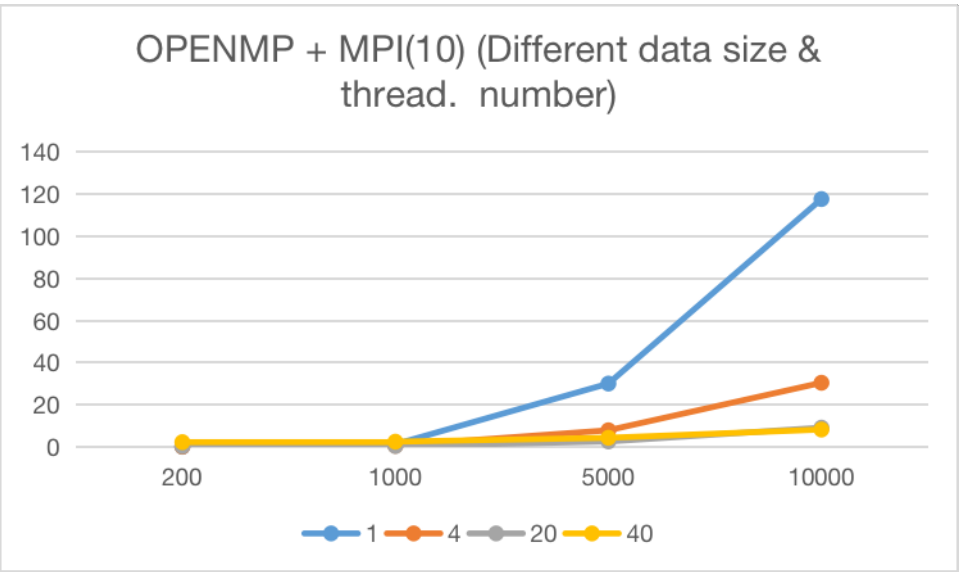

Figure 17: Time vs Thread Number – Openmp + MPI (10)

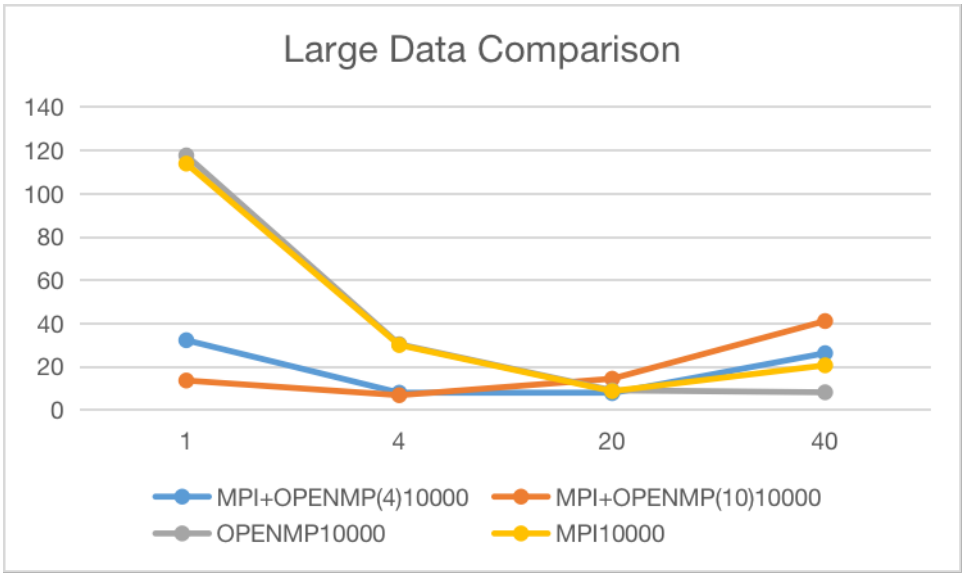

Figure 18: Time Comparison based on Large Data

Process Number	MPI+OPENMP(4)	MPI+OPENMP(10)	OPENMP	MPI
1	32.285	13.65	117.722	113.94
4	8.017	6.868	30.436	30.059
20	7.813	14.491	9.107	8.737
40	26.302	41.143	8.205	20.677

Table 9: Time Comparison based on Large Data

4 Conclusion

This report discusses three parallel mode, multi-thread (pthread, CUDA, Openmp), multi-processes (MPI), combination of multi-thread and multi-processes (MPI + openmp). We compare their performance based on different data size, process/thread number by analysing time, speedup and efficiency. In short, parallel mode can accelerate the runntime under large data size, however, the trade-off between computation and communication should be consider when choosing thread/process number.