CSC4005 Assignment 3 Report

Shiqi Yang 119010382

November 16, 2022

Introduction

In this assignment, I wrote a sequential program, a P-thread program, an CUDA program, a openMP program and an MPI program to simulate an astronomical N-body system, but in two-dimensions. The bodies are initially at rest. Their initial positions and masses are to be selected randomly (using a random number generator). The gravity between N-body should be described by the following:

$$F = G\frac{m_1 \times m_2}{r^2}$$

Also I considered the collision and bouncing, otherwise, all the points will be collapsed into a singular point. I will display the movement of the bodies using xlib. The bodies move hewing the physics formulas:

$$a = F/m$$

$$v^{'} = v + a\Delta t$$

$$x^{'} = x + v\Delta t$$

Method

I use the velocity to update the position. And the velocity is affected by gravity forces and collisions. The details are as following:

```
void update_velocity(double *m, double *m_total, double *x, double *y, double *x_total, double *y_total, double *vx, double *vy, int n) {
    // calculate force and acceleration, update velocity
    double f_x;
    double f_y;
    double _f;
    double a_x;
    double a_y;
    double d_x;
    double d_y;
    double d_square;

    for(int i = 0; i < n; i++){
        f_x = 0;
        f_y = 0;
        // calculate the distance and then accumlate the forces and update velocity if collided
        for(int j = 0; j < n; j++){
            if(i == j) break;
            d_x = x[j] - x[i];
            d_y = y[j] - y[i];
            d_square = pow(d_x , 2) + pow(d_y , 2);
            if (radius2 < d_square)
            //no collision
            {
                _f = (gravity_const * m[i] * m[j]) / (d_square + err);
                f_x += _f * d_x / sqrt(d_square);
                f_y += _f * d_y / sqrt(d_square);
            }else{
                vx[i] = -vx[i];
                vy[i] = -vy[i];
            }
```

```
        }

        // collided with the boundary
        if((x[i] + sqrt(radius2) >= bound_x && vx[i] > 0) || (x[i] - sqrt(radius2) <= 0 && vx[i] < 0)){
            vx[i] = -vx[i];
        }
        if((y[i] + sqrt(radius2) >= bound_y && vy[i] > 0) || (y[i] - sqrt(radius2) <= 0 && vy[i] < 0)){
            vy[i] = -vy[i];
        }

        // update the velocity
        a_x = f_x / m[i];
        a_y = f_y / m[i];
        vx[i] += a_x * dt;
        vy[i] += a_y * dt;
    }
}

void update_position(double *m, double *x, double *y, double *vx, double *vy, int n) {
    // update position
    for(int i = 0; i < n; i++){
        x[i] += vx[i] * dt;
        y[i] += vy[i] * dt;
    }
}
```

Sequential

As mentioned before, sequential version implementations are just updates velocity and position of each body in each iteration.

MPI

In the implementation of the MPI version, tasks are calculated by differences process. There are three parts:

- Data generation
- Data distribution
- Calculation

Details can be found in the figure 1.

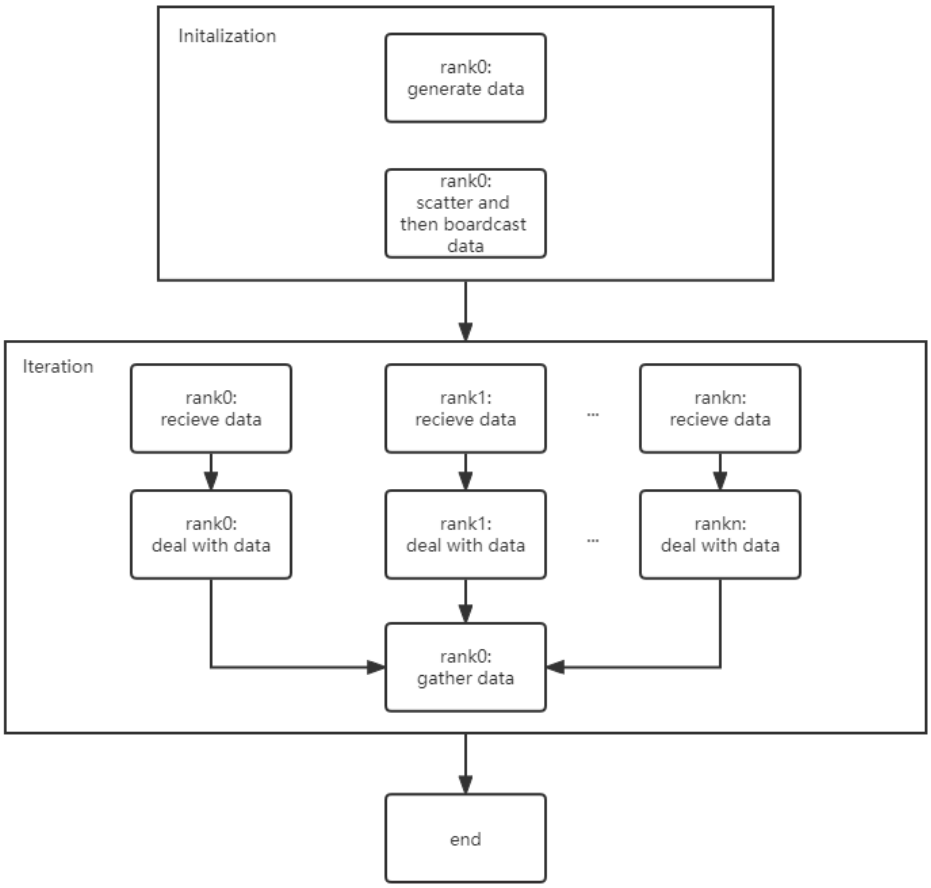


Figure 1: MPI Structure

pthread

In the pthread version, different threads of a process are used to perform calculations. Meanwhile. Figure 2 shows the implementation structure of the pthread version. In each process, a portion of the object's data is updated in each process iteration, and all threads will join at the end of each iteration.

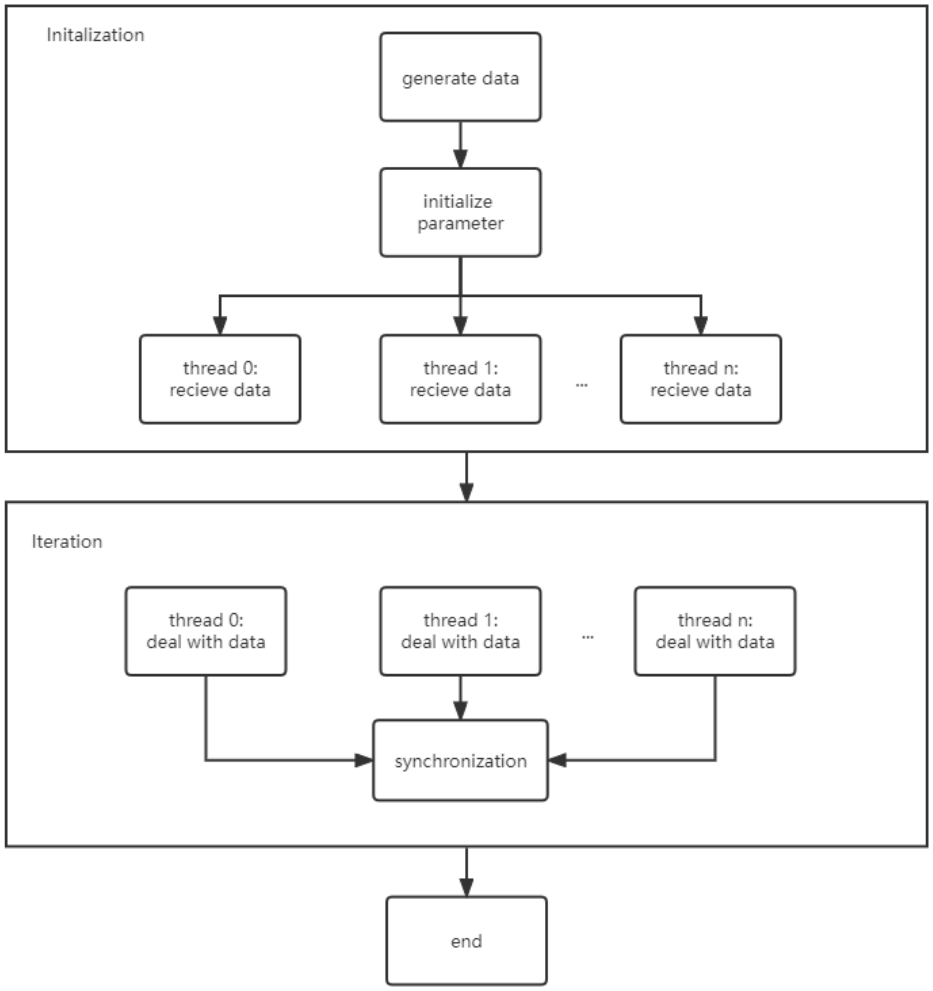


Figure 2: pthread Structure

openMP

In addition, different threads are used for calculations in the implementation of the openMP version. Figure 3 shows the implementation structure of the openMP version. Schedule multiple threads for calculations.

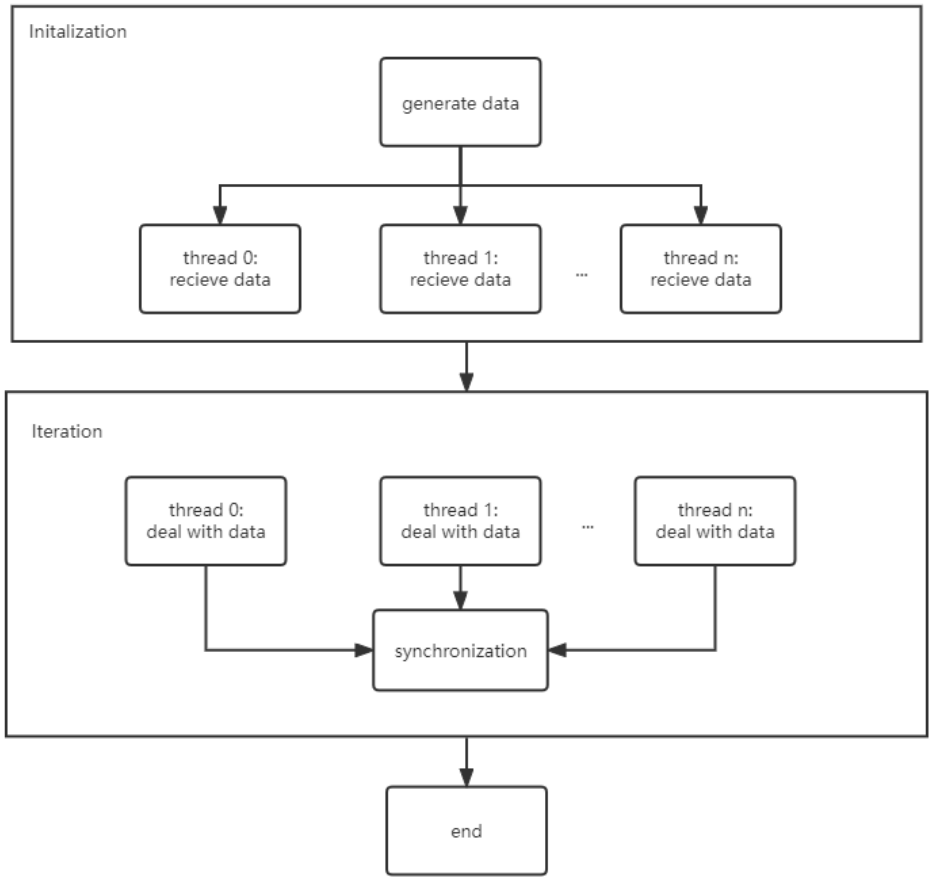


Figure 3: openMP Structure

CUDA

In the CUDA version of the implementation, computation tasks are assigned to different CUDA threads. Figure 4 shows the implementation structure of the CUDA version.

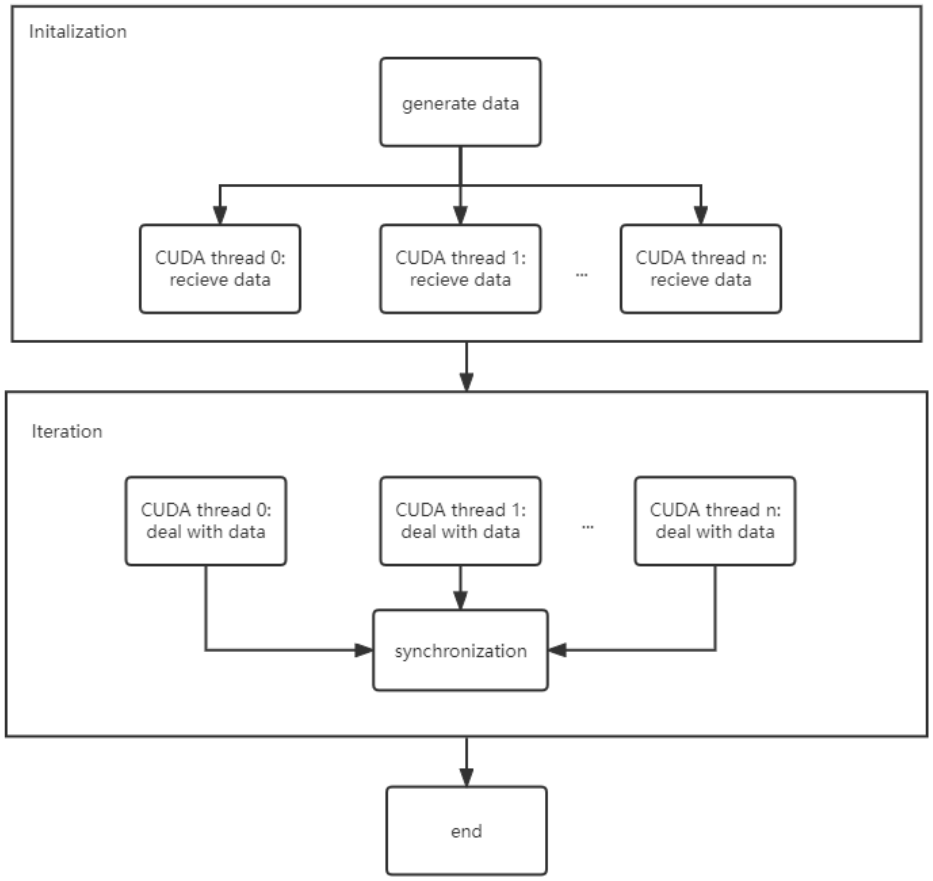


Figure 4: CUDA Structure

How to Run

Sequential

Compile

```
//Sequential (command line application):
g++ ./src/sequential.cpp -o seq -O2 -std=c++11

//Sequential (GUI application):
g++ ./src/sequential.cpp -o seqg -I/usr/include -L/usr/local/lib -L/usr/lib -lglut -lGLU -lGL -lm -DGUI -O2 -std=c++11

//MPI (command line application):
mpic++ ./src/mpi.cpp -o mpi -std=c++11

//MPI (GUI application):
mpic++ ./src/mpi.cpp -o mpig -I/usr/include -L/usr/local/lib -L/usr/lib -lglut -lGLU -lGL -lm -DGUI -std=c++11

//Pthread (command line application):
g++ ./src/pthread.cpp -o pthread -lpthread -O2 -std=c++11

//Pthread (GUI application):
g++ ./src/pthread.cpp -o pthreadg -I/usr/include -L/usr/local/lib -L/usr/lib -lglut -lGLU -lGL -lm -lpthread -DGUI -O2 -std=c++11

//CUDA (command line application): notice that nvcc is not available on VM, please use cluster.
nvcc ./src/cuda.cu -o cuda -O2 --std=c++11
```

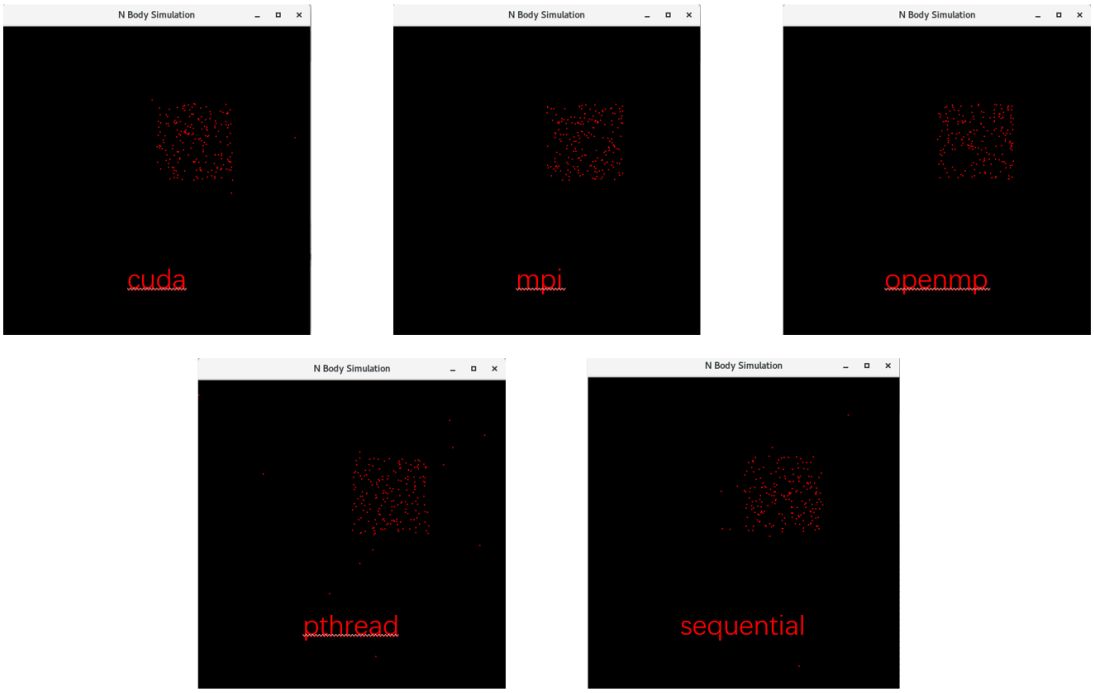
```
//CUDA (GUI application): notice that nvcc is not available on VM, please use cluster.
nvcc ./src/cuda.cu -o cudag -I/usr/include -L/usr/local/lib -L/usr/lib -lglut -lGLU -lGL -lm -O2 -DGUI --std=c++11

//OpenMP (command line application):
g++ ./src/openmp.cpp -o openmp -fopenmp -O2 -std=c++11

//OpenMP (GUI application):
g++ ./src/openmp.cpp -o openmpg -fopenmp -I/usr/include -L/usr/local/lib -L/usr/lib -lglut -lGLU -lGL -lm -O2 -DGUI -std=c++11
```

Run

```
//Sequential (command line mode):
./seq $n_body $n_iterations

//Sequential (GUI mode): please run this on VM (with GUI desktop).
./seqg $n_body $n_iterations

//MPI (command line mode):
mpirun -np $n_processes ./mpi $n_body $n_iterations

//MPI (GUI mode): please run this on VM (with GUI desktop).
mpirun -np $n_processes ./mpig $n_body $n_iterations

//Pthread (command line mode):
./pthread $n_body $n_iterations $n_threads

//Pthread (GUI mode): please run this on VM (with GUI desktop).
./pthreadg $n_body $n_iterations $n_threads

//CUDA (command line mode): for VM users, please run this on cluster.
./cuda $n_body $n_iterations

//CUDA (GUI mode): if you have both nvcc and GUI desktop, you can try this.
./cuda $n_body $n_iterations

//OpenMP (command line mode):
openmp $n_body $n_iterations $n_omp_threads

//OpenMP (GUI mode):
openmpg $n_body $n_iterations $n_omp_threads
```

Result

Image



Sequential

The time increases as the number of bodies increases.

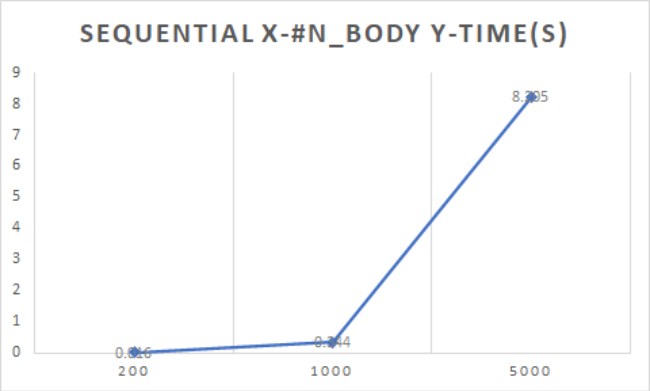


Figure 5: Sequential Result

MPI

As shown in the figure, as the number of processes increases, the elapsed time tends to decrease first and then increase.

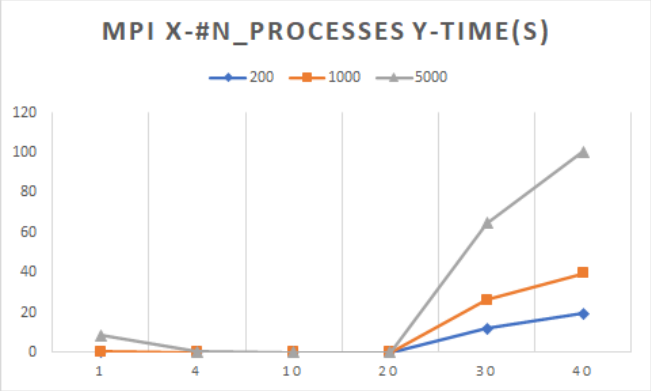


Figure 6: MPI Result

pthread

As shown in the figure, as the number of processes increases, the elapsed time tends to decrease.

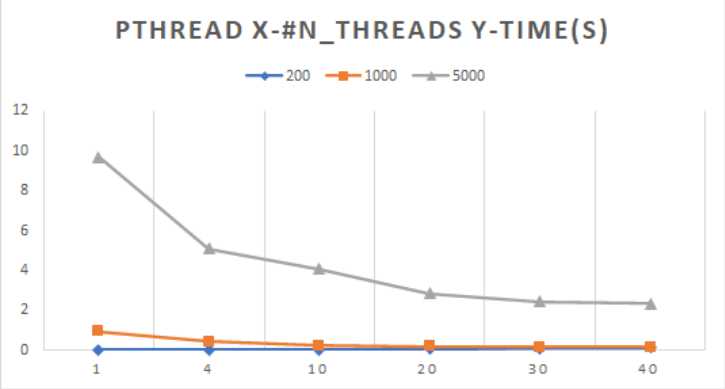


Figure 7: pthread Result

openMP

As shown in the figure, as the number of processes increases, the elapsed time tends to decrease then increase then decrease.

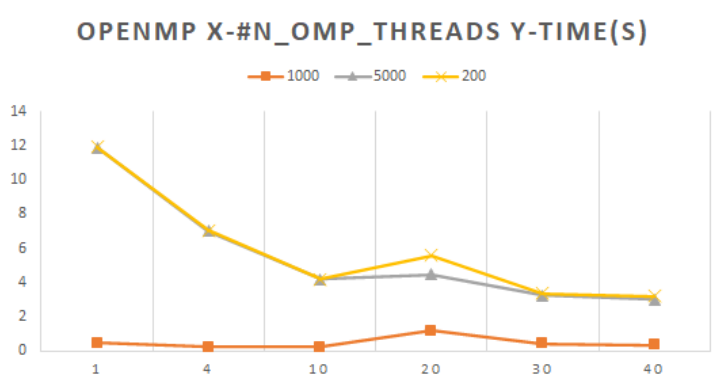


Figure 8: openMP Result

CUDA

The time increases as the number of bodies increases.

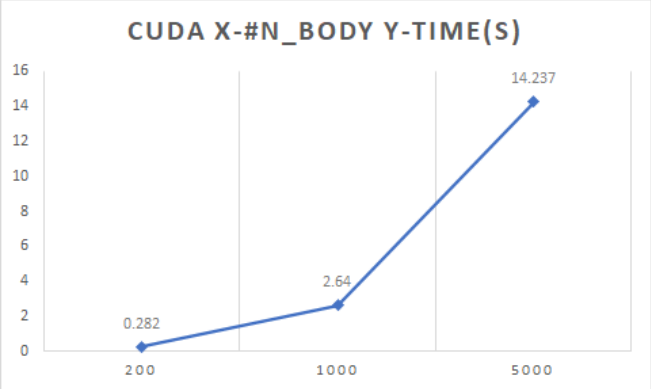


Figure 9: CUDA Result

Analysis

Comparison

I choose the best performance of each method when the body number is the same and get the following figure. We can get that, if we optimize each method, then at the most time: MPI > CUDA > OpenMP > pthread > Sequential.

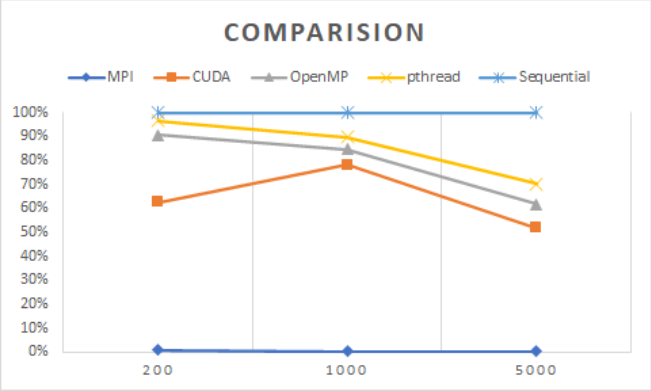


Figure 10: Comparison Result

MPI

The reason why the time spent tends to decrease is that as the process increase, the tasks handled by each process become easier and easier. This results in a reduction in the time it takes for each process to complete computation. When the number of corpses is relatively small, with simple calculations, communication between each thread becomes more consuming for the total time consumption. There is some additional waiting time when some processes that completed the calculation earlier. When the number of subjects increases, the consumption increases as the number of processes increases.

openMP

When the number of treatments is relatively small, the time consumed decreases as the number of threads increases, because the multithreading strategy saves time. When the number of threads is at a medium level, wild fluctuations occur (mainly when there are a large number of corpses). This may be due to growth fluctuations occur in quantity, the calculation is required because the body is huge. On the other hand, this

can be caused by the distribution. When the number of threads increases, it is awkward for openMP programs to process data. When the number of threads is relative, the elapsed time increases. This may be because of the increased cost of communication between threads.