香港中文大學（深圳）
The Chinese University of Hong Kong, Shenzhen

CSC4005 Assignment3 Report

Jiawei Lian

119030043

Contents

1 INTRODUCTION

N-body simulation is to simulate a two-dimension astronomical N-body system. The bodies are initialized with randomly assigned position and mass and zero velocity. The force between the bodies are calculated by

$$F = G \times \frac{m_1 \times m_2}{r^2} \tag{1}$$

Every body needs to be checked for an iteration. If a collision happens between bodies, the velocity will be reversed. If no collision happens, the acceleration will be calculated according to the Newton's law, and the velocity will be updated according to the acceleration. In this project, I implemented a sequential program, a MPI program, a P-thread program, a CUDA program, an openMP program, and a MPI+OpenMP program to do the N-body simulation.

2 METHOD

2.1 DESIGN OF THE PROGRAM

2.1.1 DESIGN OF THE PARTITION METHOD

The size of the output figure and the number of processes or threads used in the program are different. To ensure the program in robust enough to work for different settings, I used the formulae (use process as an example)

$$part\_body = ((size - rank)\%world\_size > 0) + (size - rank)/world\_size \tag{2}$$

to calculate the **part_body** of each process or thread. Here, I compare each body with each other bodies. **world_size** is the number of total processes or threads. For MPI version, **rank** is the rank of the current process. For Pthread version, **rank** is the index of the current thread. Figure 1 illustrate the idea of the partition method.

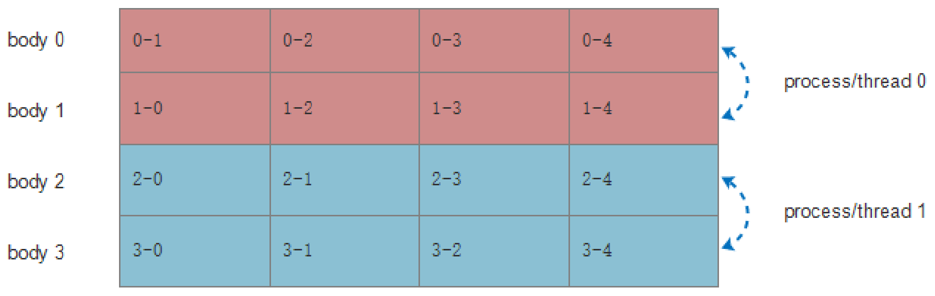
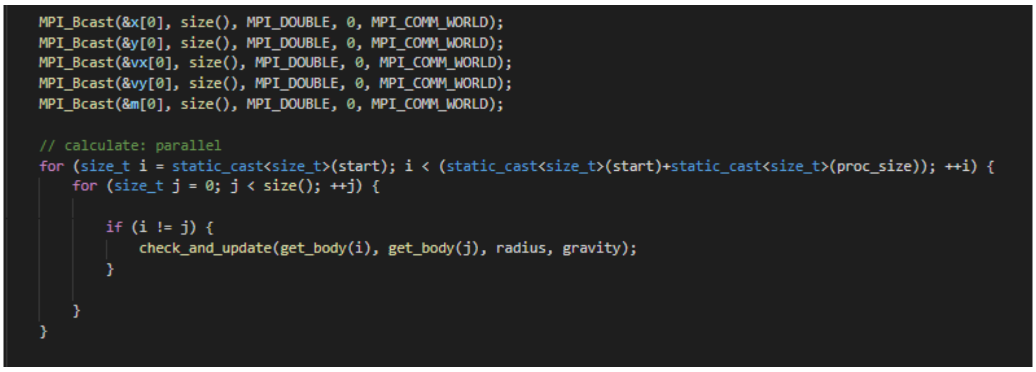


FIGURE 1: THE PARTITION METHOD EXAMPLE

2.1.2 DESIGN OF THE MPI VERSION

First the program will take the input of the number of bodies **bodies**. Then, the process with rank 0 will send the information about the pool to each process. Each process will calculate its **part_bodies** according to the formulae (2). Then, the process with rank 0 will Gather information about the **part_bodies** of each process. Next, the process with rank 0 will calculate the start position **start** of each process and scatter. After receiving the start position, each process will start the calculation. In the pool, apart from the original vectors **x**, **y**, **vx**, **vy**, **ax**, **ay** that stores the current position, velocity, and acceleration of the bodies, I created another set of vectors, **dx**, **dy**, **dvx**, **dvy**, **dax**, **day** that store the changes made in each comparison. At first, the root will broadcast the original vectors used as snapshot. Then each process will first initialize the **ax**, **ay**, **dx**, **dy**, **dvx**, **dvy**, **dax**, **day** to zero, then check the collision condition. If collision happens, the values in **dvx**, **dvy** of that body will be changed. If no collision happens, the values in **dax**, **day** will be changed. After each process finishes the calculation, they will send their **dx**, **dy**, **dvx**, **dvy**, **dax**, **day** back to the root. The root will sum up the changes of one body and add the total change to the original vectors. Figure 2 show the broadcast of the snapshot. Figure 3 shows the **check_and_update**. Figure 4 shows the addition of the changes.

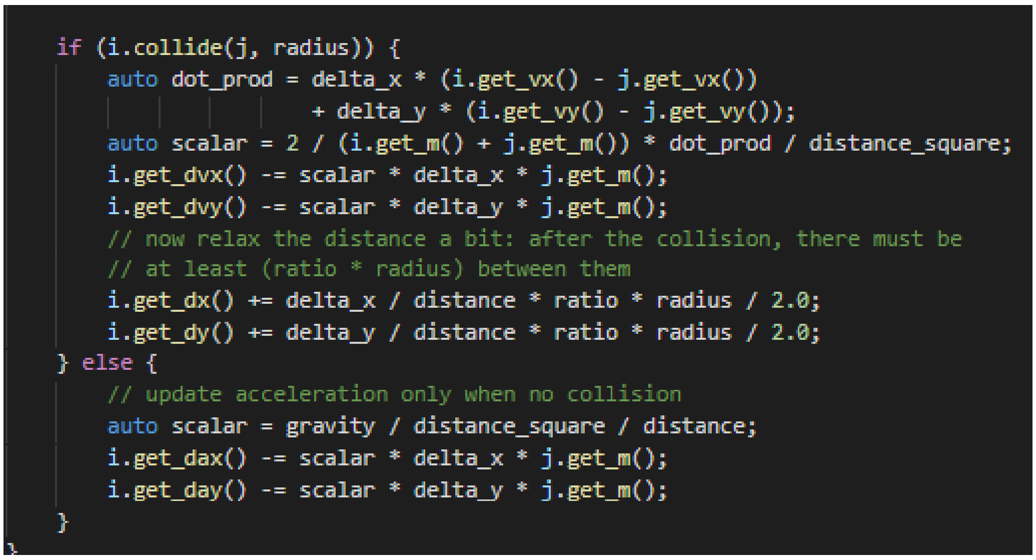


```
MPI_Bcast(&x[0], size(), MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&y[0], size(), MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&vx[0], size(), MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&vy[0], size(), MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&m[0], size(), MPI_DOUBLE, 0, MPI_COMM_WORLD);

// calculate: parallel
for (size_t i = static_cast<size_t>(start); i < (static_cast<size_t>(start)+static_cast<size_t>(proc_size)); ++i) {
    for (size_t j = 0; j < size(); ++j) {

        if (i != j) {
            check_and_update(get_body(i), get_body(j), radius, gravity);
        }

    }
}
```

FIGURE 2: BROADCAST OF THE SNAPSHOT

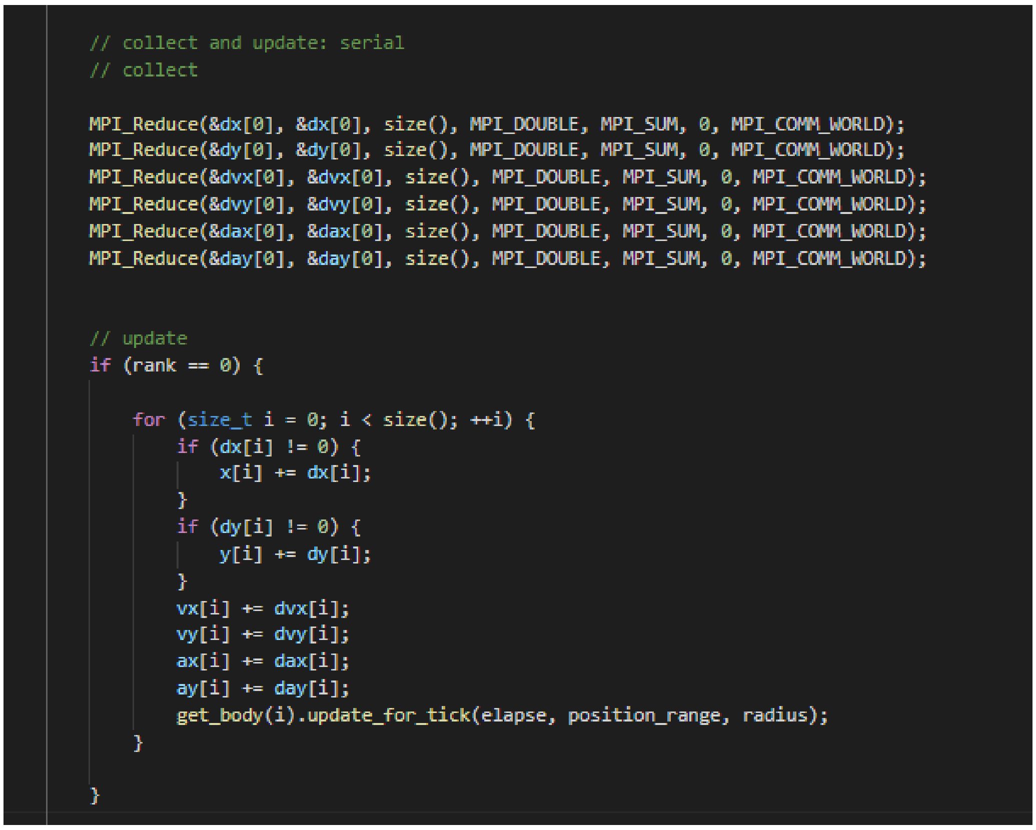
2.1.3 DESIGN OF THE PTHREAD VERSION AND OPENMP VERSION

First the program will take the input of the **num_of_thread** and **bodies**. If **num_of_thread** is larger than **bodies**, then **num_of_thread** will be set to be equal to **bodies**. As threads share the memory, there is no need to passing messages. Then the program will calculate the **part_bodies** and **start** of each thread by using the formulae (2), then create the threads. Each thread will first check the collision, and reverse the velocity if there is a collision, or calculate the total force if there is no collision. The idea of **check_and_update** is similar to the MPI version. Then to ensure that there is no data racing, synchronization technology will be used. In the pool, vector **x**, **y**, **vx**, **vy**, **ax**, **ay** stores the current position, velocity, acceleration, and mass of each body. As these vectors are shared among the threads, we create another sets of vectors **dx**, **dy**, **dvx**, **dvy**,



```
if (i.collide(j, radius)) {
    auto dot_prod = delta_x * (i.get_vx() - j.get_vx())
                    + delta_y * (i.get_vy() - j.get_vy());
    auto scalar = 2 / (i.get_m() + j.get_m()) * dot_prod / distance_square;
    i.get_dvx() -= scalar * delta_x * j.get_m();
    i.get_dvy() -= scalar * delta_y * j.get_m();
    // now relax the distance a bit: after the collision, there must be
    // at least (ratio * radius) between them
    i.get_dx() += delta_x / distance * ratio * radius / 2.0;
    i.get_dy() += delta_y / distance * ratio * radius / 2.0;
} else {
    // update acceleration only when no collision
    auto scalar = gravity / distance_square / distance;
    i.get_dax() -= scalar * delta_x * j.get_m();
    i.get_day() -= scalar * delta_y * j.get_m();
}
```

FIGURE 3: CHECK AND UPDATE ONE BODY



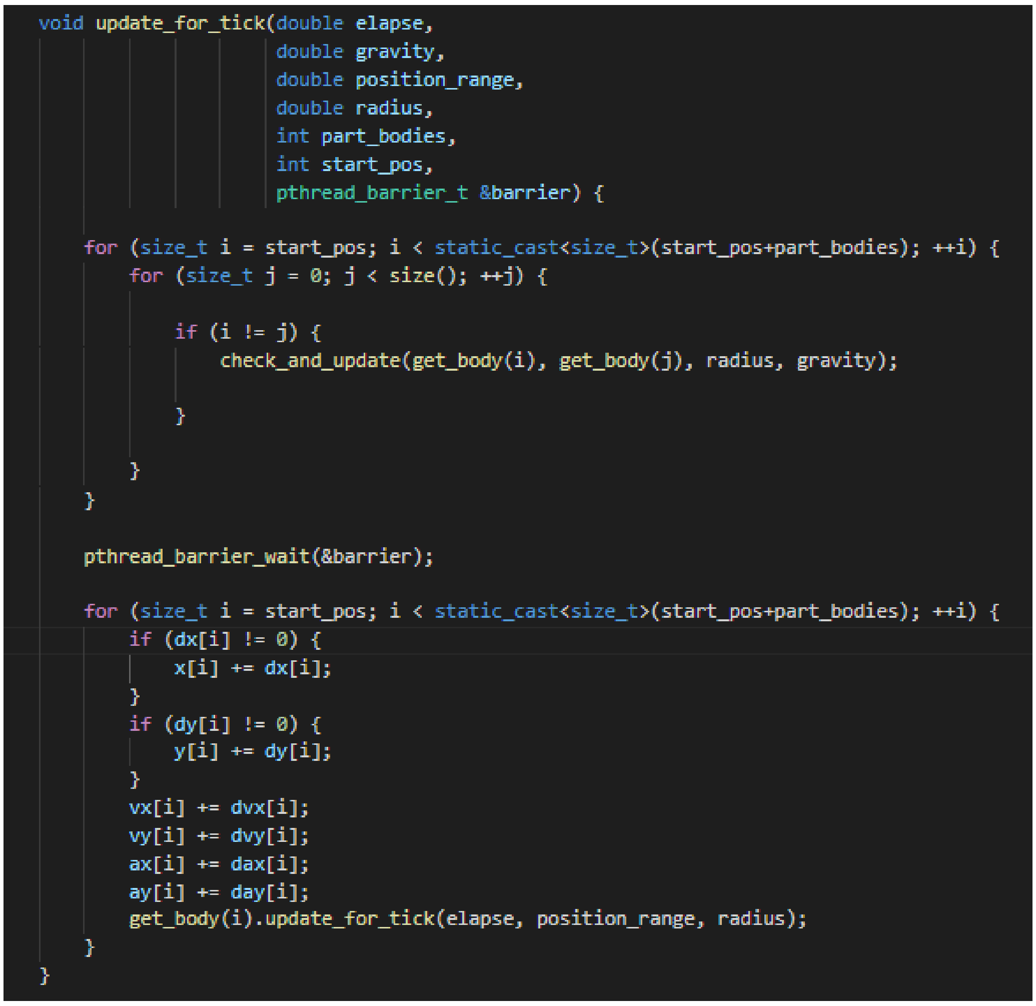
```
// collect and update: serial
// collect

MPI_Reduce(&dx[0], &dx[0], size(), MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
MPI_Reduce(&dy[0], &dy[0], size(), MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
MPI_Reduce(&dvx[0], &dvx[0], size(), MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
MPI_Reduce(&dvy[0], &dvy[0], size(), MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
MPI_Reduce(&dax[0], &dax[0], size(), MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
MPI_Reduce(&day[0], &day[0], size(), MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);


// update
if (rank == 0) {

    for (size_t i = 0; i < size(); ++i) {
        if (dx[i] != 0) {
            x[i] += dx[i];
        }
        if (dy[i] != 0) {
            y[i] += dy[i];
        }
        vx[i] += dvx[i];
        vy[i] += dvy[i];
        ax[i] += dax[i];
        ay[i] += day[i];
        get_body(i).update_for_tick(elapse, position_range, radius);
    }

}
```

FIGURE 4: ADDITION OF THE CHANGES

dax, and **day** to store the changes during the **check_and_update**. So we can avoid read-write data racing. The vectors used to store the changes are also shared among the threads. But as a thread will only update a part of the bodies, and there is no overlap, so there is no write-write conflicts in the latter set of vectors. They can write the total changes into the pool vectors in parallel for this reason, after all threads finished the **check_and_update**. After each thread completes the calculation, they will join in the main function.

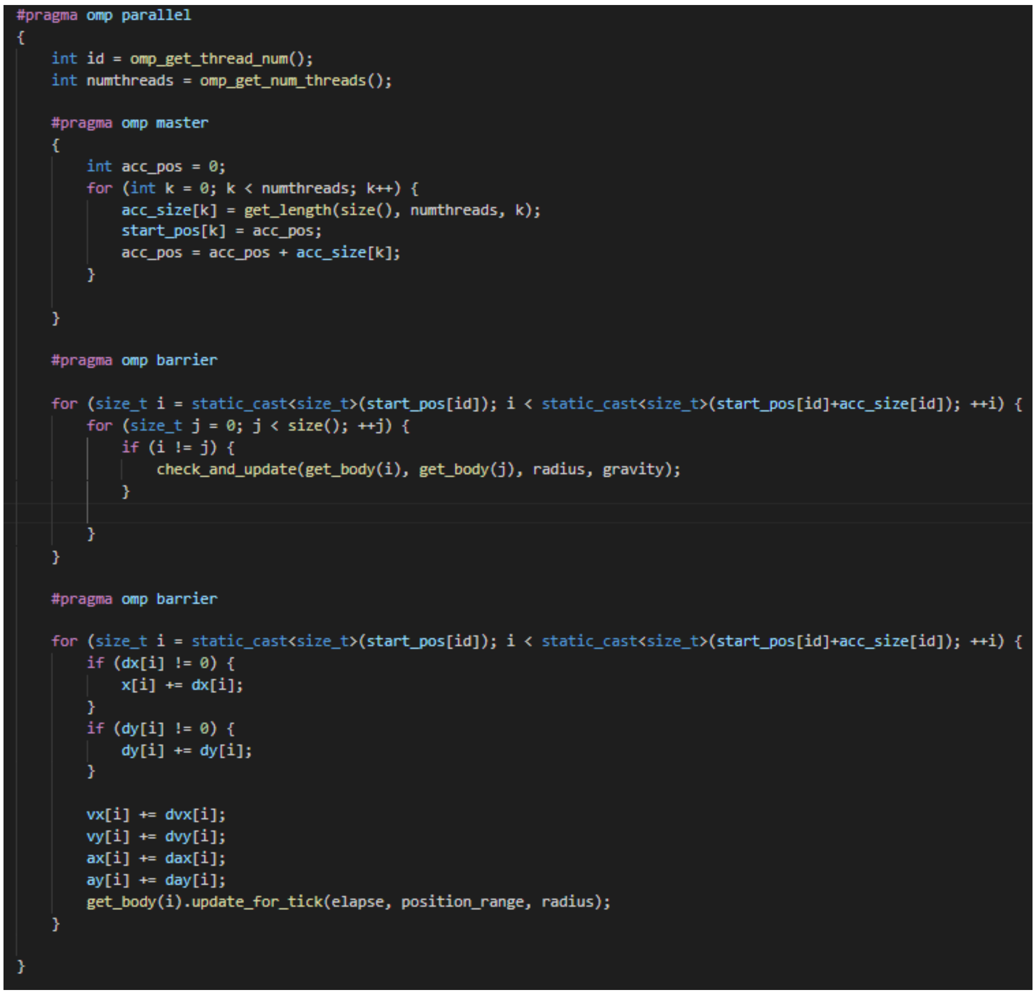
As OpenMP uses pthread, the implementation of the pthread version and the OpenMP version is quite similar.

Figure 5 shows the synchronization of threads in the pthread version. Figure 6 shows the synchronization of threads in the OpenMP version.



```
void update_for_tick(double elapse,
                     double gravity,
                     double position_range,
                     double radius,
                     int part_bodies,
                     int start_pos,
                     pthread_barrier_t &barrier) {

    for (size_t i = start_pos; i < static_cast<size_t>(start_pos+part_bodies); ++i) {
        for (size_t j = 0; j < size(); ++j) {

            if (i != j) {
                check_and_update(get_body(i), get_body(j), radius, gravity);

            }

        }
    }

    pthread_barrier_wait(&barrier);

    for (size_t i = start_pos; i < static_cast<size_t>(start_pos+part_bodies); ++i) {
        if (dx[i] != 0) {
            x[i] += dx[i];
        }
        if (dy[i] != 0) {
            y[i] += dy[i];
        }
        vx[i] += dvx[i];
        vy[i] += dvy[i];
        ax[i] += dax[i];
        ay[i] += day[i];
        get_body(i).update_for_tick(elapse, position_range, radius);
    }
}
```

FIGURE 5: ADDITION OF THE CHANGES



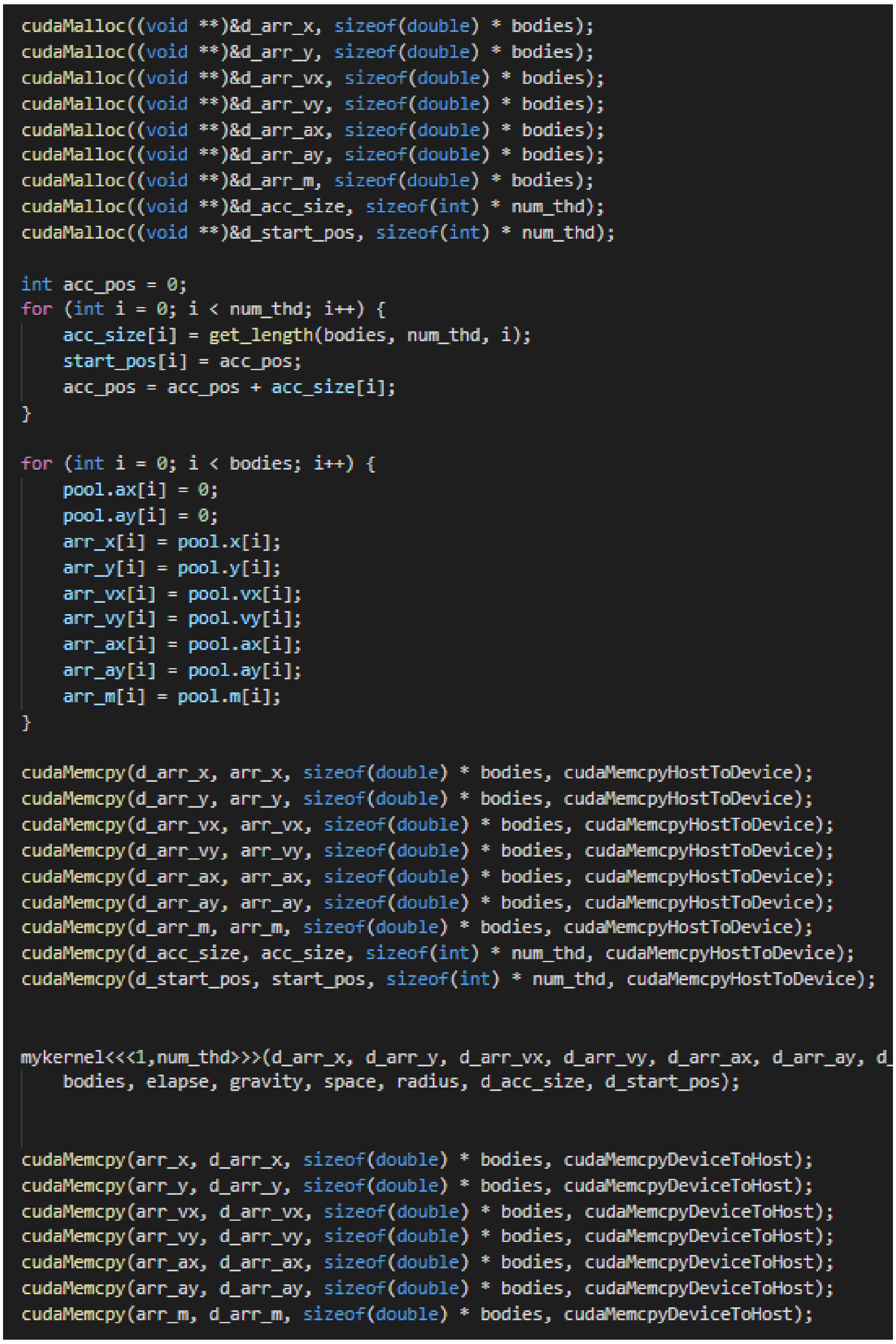
```
#pragma omp parallel
{
    int id = omp_get_thread_num();
    int numthreads = omp_get_num_threads();

    #pragma omp master
    {
        int acc_pos = 0;
        for (int k = 0; k < numthreads; k++) {
            acc_size[k] = get_length(size(), numthreads, k);
            start_pos[k] = acc_pos;
            acc_pos = acc_pos + acc_size[k];
        }

    }

    #pragma omp barrier

    for (size_t i = static_cast<size_t>(start_pos[id]); i < static_cast<size_t>(start_pos[id]+acc_size[id]); ++i) {
        for (size_t j = 0; j < size(); ++j) {
            if (i != j) {
                check_and_update(get_body(i), get_body(j), radius, gravity);
            }

        }
    }

    #pragma omp barrier

    for (size_t i = static_cast<size_t>(start_pos[id]); i < static_cast<size_t>(start_pos[id]+acc_size[id]); ++i) {
        if (dx[i] != 0) {
            x[i] += dx[i];
        }
        if (dy[i] != 0) {
            dy[i] += dy[i];
        }

        vx[i] += dvx[i];
        vy[i] += dvy[i];
        ax[i] += dax[i];
        ay[i] += day[i];
        get_body(i).update_for_tick(elapse, position_range, radius);
    }

}
```

FIGURE 6: ADDITION OF THE CHANGES

2.1.4 DESIGN OF THE CUDA VERSION

First the host launch the program and take the input of the **num_of_thread** and **bodies**. If **num_of_thread** is larger than **bodies**, then **num_of_thread** will be set to be equal to **bodies**. Then the host will calculate the **part_bodies** and **start** of each thread by using the formulae (2). Then the host allocates GPU memory and copies the information of the pool into GPU. Next, the host launched GPU threads. Each thread will do the same calculation as in the pthread an OpenMP version. After the calculation, the host will copy the updated information from GPU to CPU, and write them back into the pool. Finally, the CPU will free the memory that it allocates at the beginning.

Figure 7 shows the memory copy and the launch of the GPU threads, and copy the results to CPU. This program only utilize one block of the GPU. Figure 5 shows the synchronization of threads in the pthread version.



```
cudaMalloc((void **)&d_arr_x, sizeof(double) * bodies);
cudaMalloc((void **)&d_arr_y, sizeof(double) * bodies);
cudaMalloc((void **)&d_arr_vx, sizeof(double) * bodies);
cudaMalloc((void **)&d_arr_vy, sizeof(double) * bodies);
cudaMalloc((void **)&d_arr_ax, sizeof(double) * bodies);
cudaMalloc((void **)&d_arr_ay, sizeof(double) * bodies);
cudaMalloc((void **)&d_arr_m, sizeof(double) * bodies);
cudaMalloc((void **)&d_acc_size, sizeof(int) * num_thd);
cudaMalloc((void **)&d_start_pos, sizeof(int) * num_thd);

int acc_pos = 0;
for (int i = 0; i < num_thd; i++) {
    acc_size[i] = get_length(bodies, num_thd, i);
    start_pos[i] = acc_pos;
    acc_pos = acc_pos + acc_size[i];
}

for (int i = 0; i < bodies; i++) {
    pool.ax[i] = 0;
    pool.ay[i] = 0;
    arr_x[i] = pool.x[i];
    arr_y[i] = pool.y[i];
    arr_vx[i] = pool.vx[i];
    arr_vy[i] = pool.vy[i];
    arr_ax[i] = pool.ax[i];
    arr_ay[i] = pool.ay[i];
    arr_m[i] = pool.m[i];
}

cudaMemcpy(d_arr_x, arr_x, sizeof(double) * bodies, cudaMemcpyHostToDevice);
cudaMemcpy(d_arr_y, arr_y, sizeof(double) * bodies, cudaMemcpyHostToDevice);
cudaMemcpy(d_arr_vx, arr_vx, sizeof(double) * bodies, cudaMemcpyHostToDevice);
cudaMemcpy(d_arr_vy, arr_vy, sizeof(double) * bodies, cudaMemcpyHostToDevice);
cudaMemcpy(d_arr_ax, arr_ax, sizeof(double) * bodies, cudaMemcpyHostToDevice);
cudaMemcpy(d_arr_ay, arr_ay, sizeof(double) * bodies, cudaMemcpyHostToDevice);
cudaMemcpy(d_arr_m, arr_m, sizeof(double) * bodies, cudaMemcpyHostToDevice);
cudaMemcpy(d_acc_size, acc_size, sizeof(int) * num_thd, cudaMemcpyHostToDevice);
cudaMemcpy(d_start_pos, start_pos, sizeof(int) * num_thd, cudaMemcpyHostToDevice);


mykernel<<<1,num_thd>>>(d_arr_x, d_arr_y, d_arr_vx, d_arr_vy, d_arr_ax, d_arr_ay, d_
    bodies, elapse, gravity, space, radius, d_acc_size, d_start_pos);


cudaMemcpy(arr_x, d_arr_x, sizeof(double) * bodies, cudaMemcpyDeviceToHost);
cudaMemcpy(arr_y, d_arr_y, sizeof(double) * bodies, cudaMemcpyDeviceToHost);
cudaMemcpy(arr_vx, d_arr_vx, sizeof(double) * bodies, cudaMemcpyDeviceToHost);
cudaMemcpy(arr_vy, d_arr_vy, sizeof(double) * bodies, cudaMemcpyDeviceToHost);
cudaMemcpy(arr_ax, d_arr_ax, sizeof(double) * bodies, cudaMemcpyDeviceToHost);
cudaMemcpy(arr_ay, d_arr_ay, sizeof(double) * bodies, cudaMemcpyDeviceToHost);
cudaMemcpy(arr_m, d_arr_m, sizeof(double) * bodies, cudaMemcpyDeviceToHost);
```

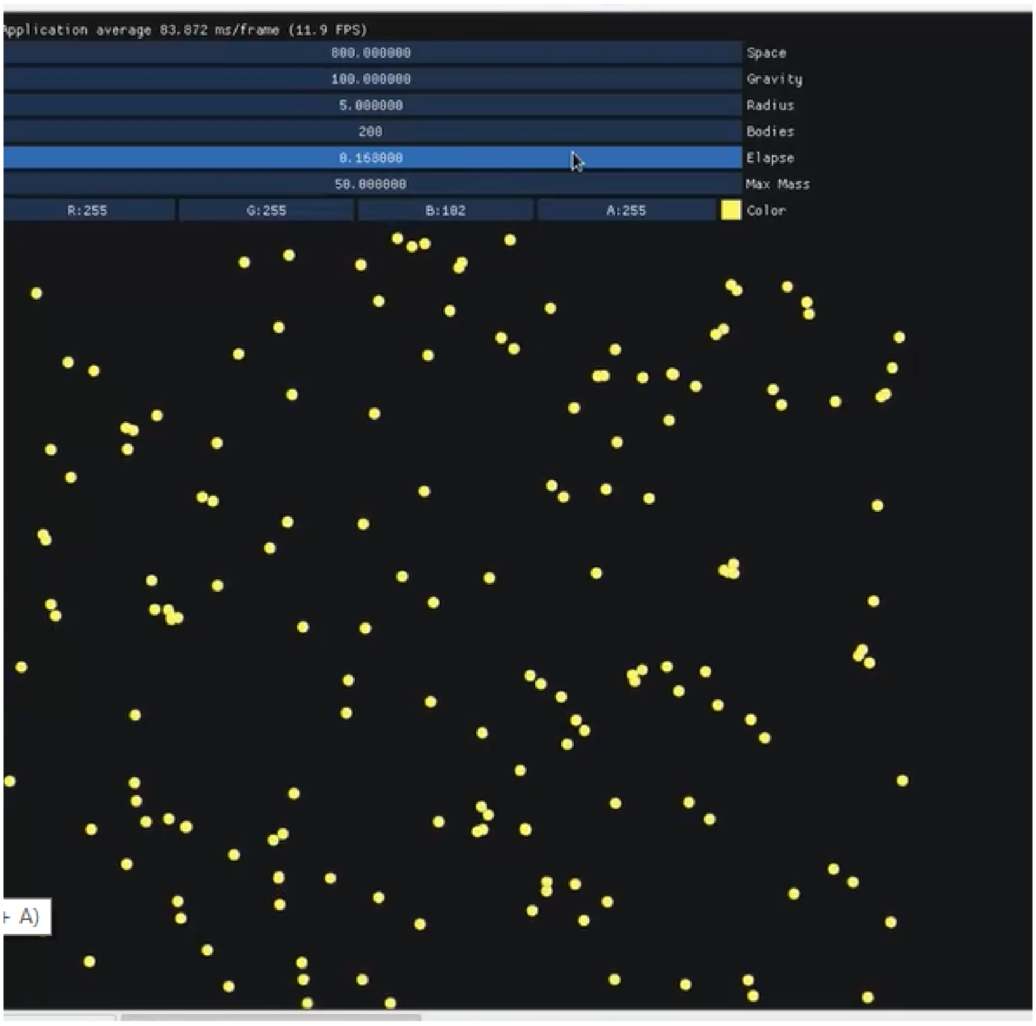
FIGURE 7: CUDA MEMORY MANAGEMENT

2.1.5 DESIGN OF THE MPI+OPENMP VERSION

First the program will take the input of the **bodies**. Then, the process with rank 0 will send the information about the pool, specifically, **bodies**, **space**, **max_mass**, **elapse**, **gravity**, and **radius**, to each other process. Each process will calculate **part_bodies**, the bodies that it controls, according to the formulae. Then, the process with rank 0 will Gather information about the

part_bodies of each process. Next, the process with rank 0 will calculate the start position **start** of each process and scatter. The start position **start** together with **part_bodies** indicate the index of the bodies that each process controls. After receiving the start position, each process will start forking threads. The process will calculate the number of bodies that each thread controls. The number is calculated by the formulae (2). And the start index is the start index of the body that the process control. Then each thread does the same work as the thread in the original OpenMP version. After the calculation of each thread is done, each process send the changes of the bodies that it controls to the root. The root will sum these changes up, and add this sum to the corresponding body.

2.1.6 SNAPSHOT OF THE OUTPUT



```
pplication average 83.872 ms/frame (11.9 FPS)
800.000000                                   Space
100.000000                                   Gravity
5.000000                                     Radius
200                                          Bodies
0.168000                                     Elapse
50.000000                                    Max Mass
R:255      G:255      B:182      A:255       Color
```

FIGURE 8: SNAPSHOT

The flowcharts of the MPI version, the pthread version and OpenMP version, the cuda version, and the MPI+OpenMP version, are attached on the last four pages.

2.1.7 THE EXPERIMENT PIPELINE

Write python scripts to generate sbatch shell scripts with different number of processes or threads, and different figure size. Then write a shell script to submit the sbatch shell scripts. Take the average of the first five values of **number of bodies per second** to represent speed. Finally, use R to visualize the results.

The flowcharts of the programs are attached on the last pages.

3 RESULTS

3.1 MPI: THE NUMBER OF PROCESSES USED IN THE PROGRAM

In this section, we examine the effect of the number of processes used in the program. We can observe that figure size affects the impact of core numbers on speed:

1. In small and medium number of bodies cases (bodies_num < 300): the speed increases first then decrease as the core number increases.

2. In large number of bodies cases (bodies_num >= 1000): the speed increases as the process number increases. Some decreases appear at process number 32, 64, 96, 128)

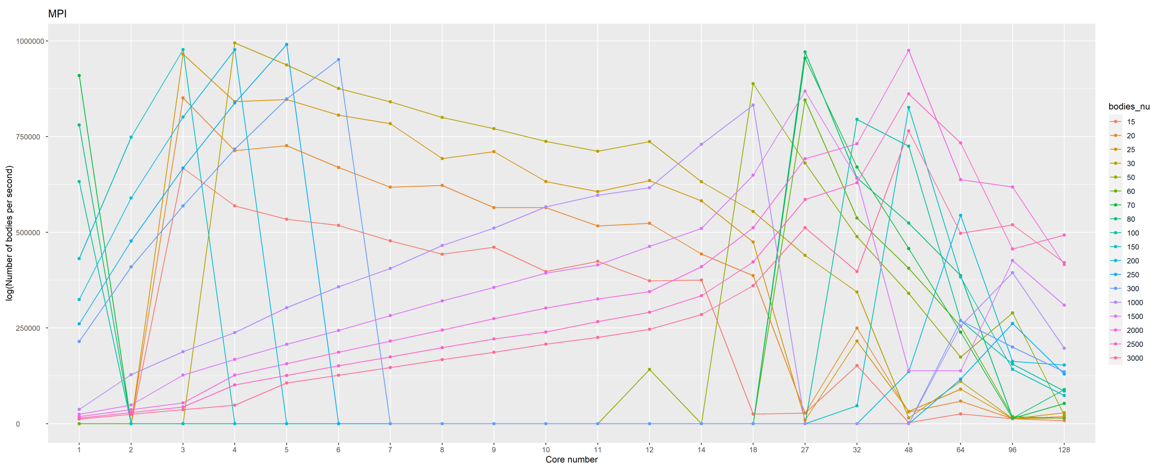


FIGURE 9: CORE_NUM-SPEED RELATIONSHIP

3.2 PTHREAD: THE NUMBER OF THREADS USED IN THE PROGRAM

In this section, we examine the effect of the number of threads used in the program. From the figure, we can see that generally speaking, the speed does not get any faster when the number of thread increases. When the number of thread is greater than 32, the speed shrinks faster.

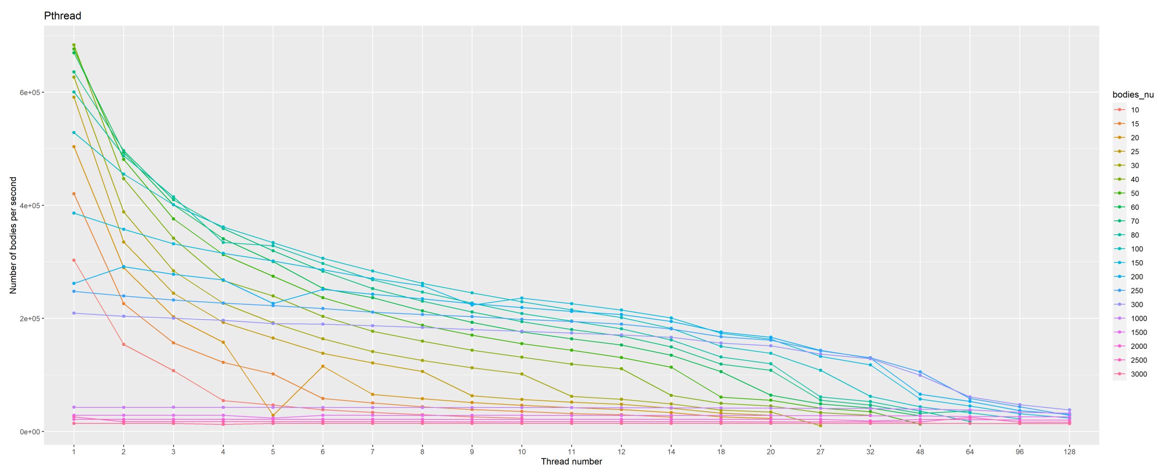


FIGURE 10: THREAD_NUM-SPEED RELATIONSHIP

3.3 OPENMP: THE NUMBER OF THREADS USED IN THE PROGRAM

In this section, we examine the effect of the number of threads used in the program. From the figure, we can see that generally speaking, the speed does not get any faster when the number of thread increases.

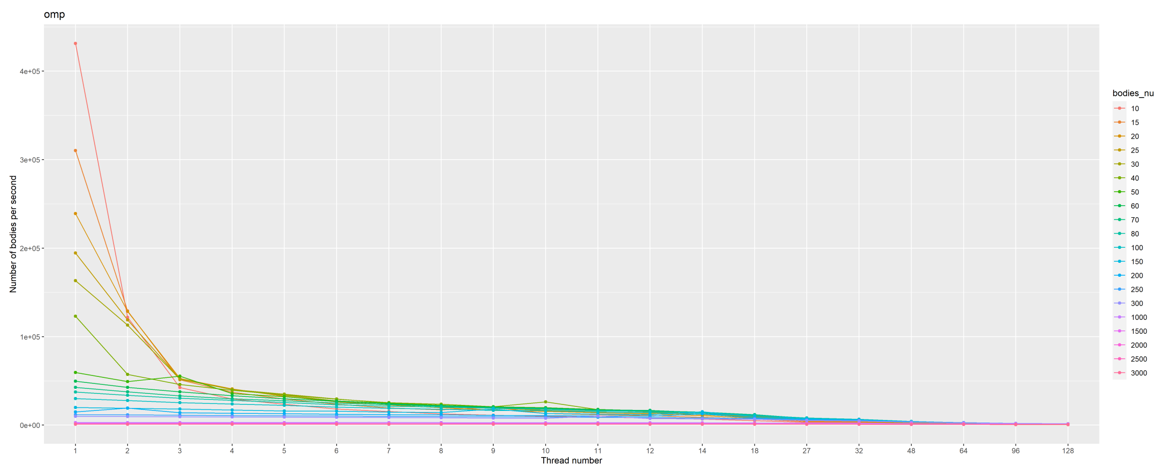


FIGURE 11: THREAD_NUM-SPEED RELATIONSHIP

3.4 CUDA: THE NUMBER OF THREADS USED IN THE PROGRAM

In this section, we examine the effect of the number of threads used in the program. From the figure, we can see that generally speaking, the speed gets faster when the number of thread increases.

3.5 MPI+OPENMP: THE NUMBER OF NODES USED IN THE PROGRAM

In this section, we examine the effect of the number of nodes used in the program. In the experiment, we fixed the total thread number to be 96 and change the node number. From

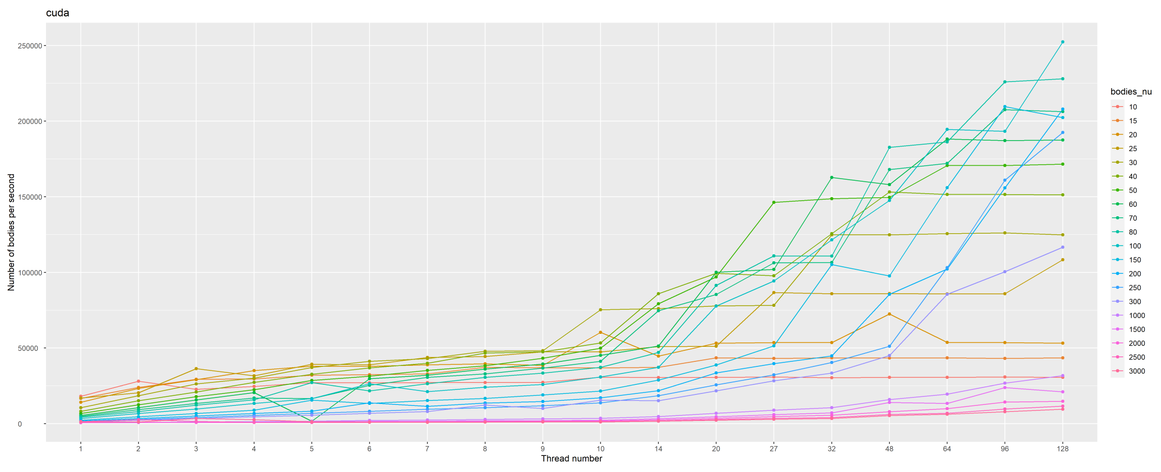


FIGURE 12: THREAD_NUM-SPEED RELATIONSHIP

the figure, we can see that when node == 1, the speed is the fastest. When n == 2, the speed get much slower. When node == 3, the speed gets faster than when n == 2, but no faster than node == 1.

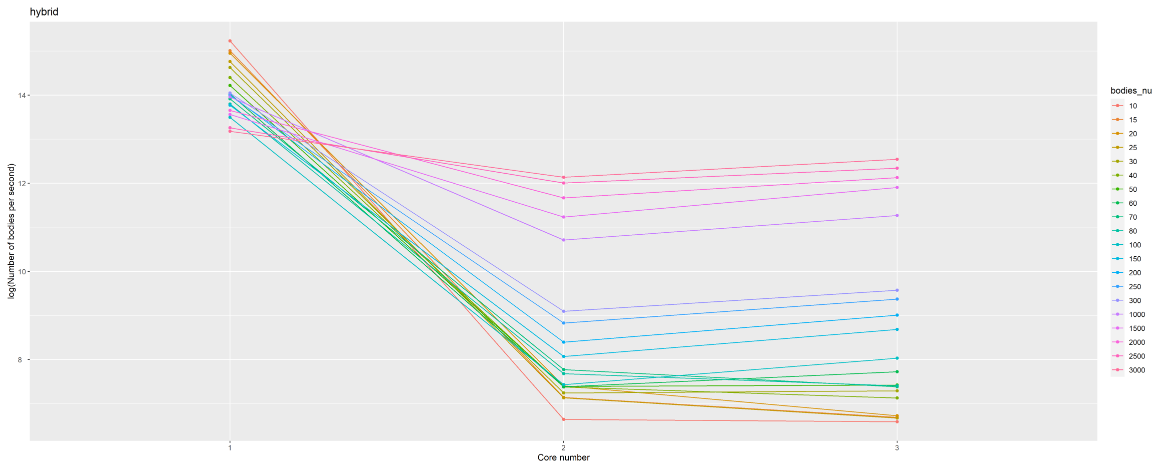


FIGURE 13: NODE_NUM-SPEED RELATIONSHIP

3.6 MPI VS SEQUENTIAL VS PTHREAD VS OPENMP VS CUDA

We use the MPI version with core = 1 to represent the performance of the Sequential version. We can observe that:

1. The sequential version and the MPI version performs good when the number of bodies is large.
2. Pthread performs better than OpenMP
3. CUDA version performs best when the number of thread or the number of process is large.

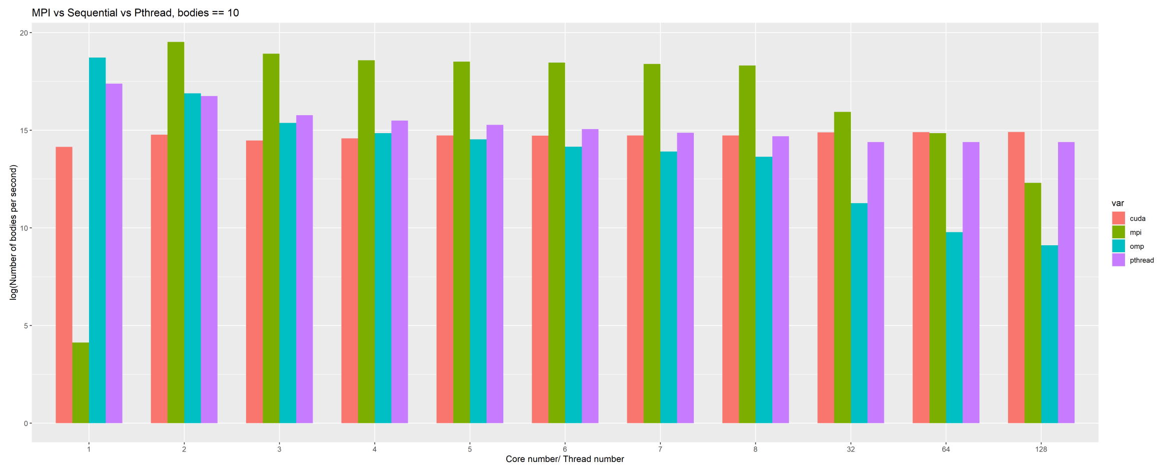


FIGURE 14: COMPARE WHEN FIGURE SIZE == 10

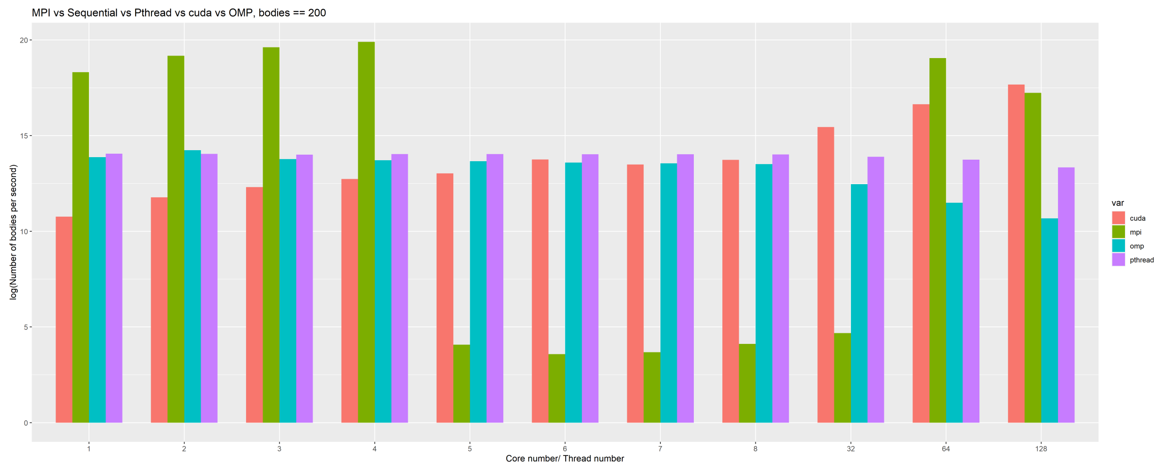


FIGURE 15: COMPARE WHEN FIGURE SIZE == 200

3.7 SIZE OF THE BODIES

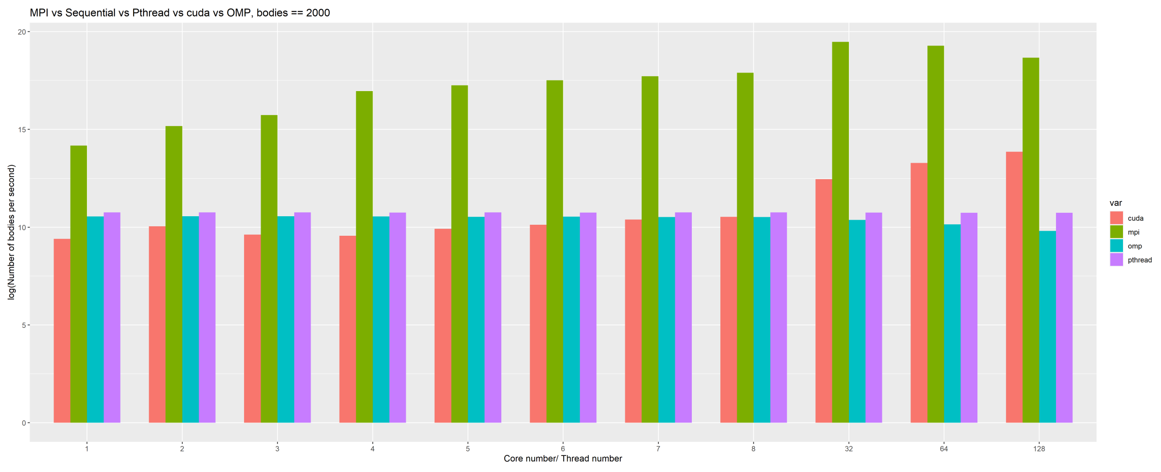


FIGURE 16: COMPARE WHEN FIGURE SIZE == 2000

3.7.1 MPI VERSION

1. When use one node, the speed first increases then decreases, and finally increases, as the body number increases.
2. When use multiple nodes, the speed first increases, and then decreases

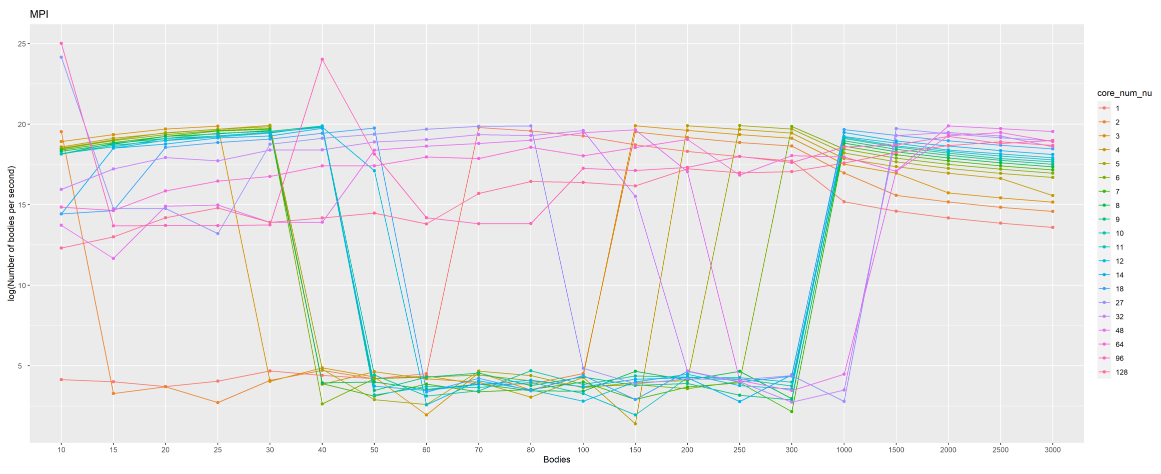


FIGURE 17: MPI: BODIES-SPEED RELATIONSHIP

3.7.2 PTHREAD VERSION

1. The speed first slightly increases, then decreases as the body number increases.

3.7.3 OPENMP VERSION

1. The speed first slightly increases, then decreases as the body number increases.

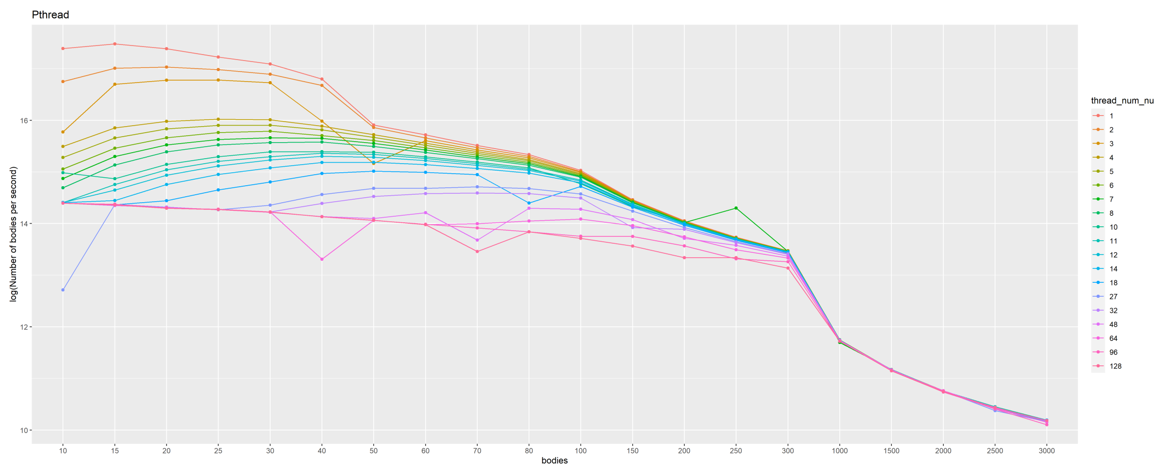


FIGURE 18: PTHREAD: BODIES-SPEED RELATIONSHIP

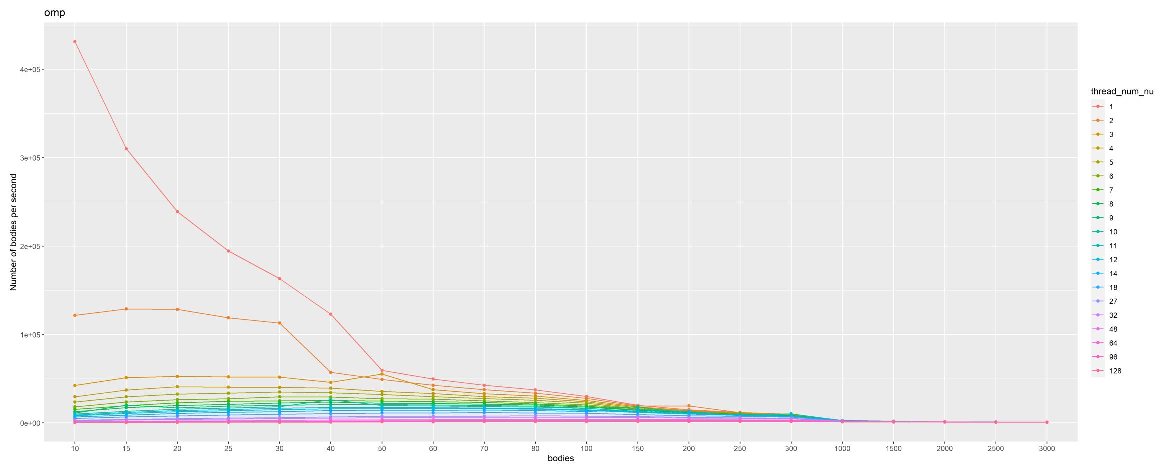


FIGURE 19: OPENMP: BODIES-SPEED RELATIONSHIP

3.7.4 CUDA VERSION

1. The speed first slightly increases, then decreases as the body number increases.

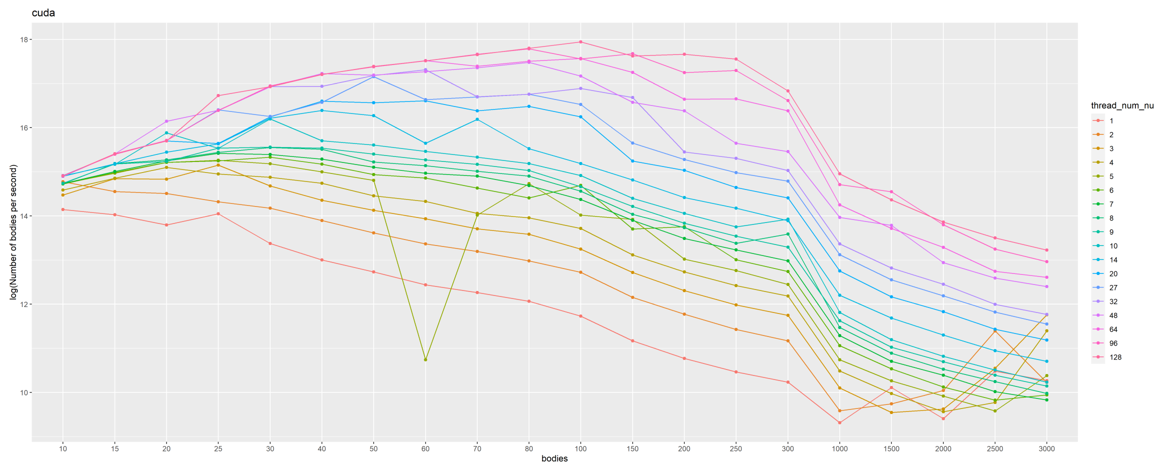


FIGURE 20: CUDA: BODIES-SPEED RELATIONSHIP

3.7.5 MPI+OPENMP VERSION

1. When the node == 1: the speed keeps decreasing.
2. When the node == 2 and node == 3: the speed first slightly increases, then decreases as the body number increases.

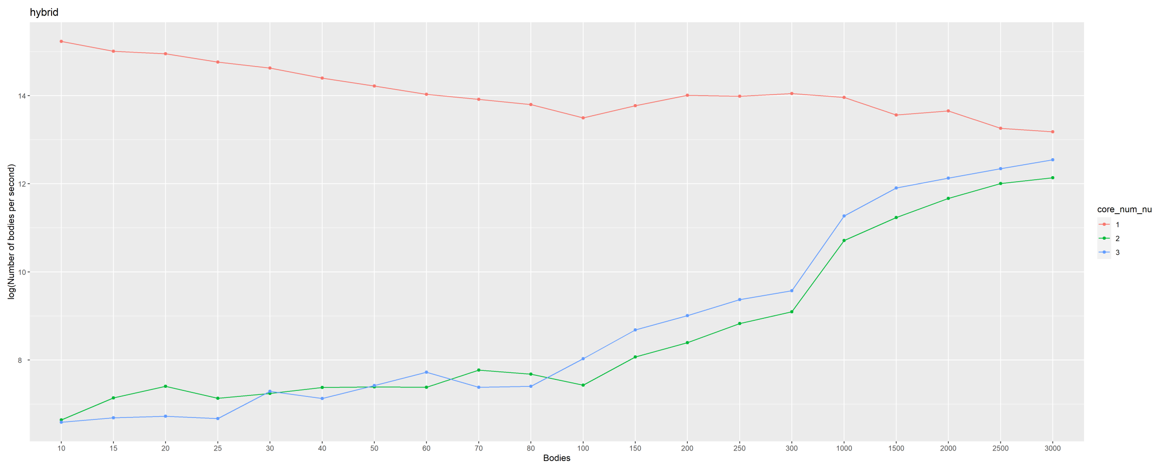


FIGURE 21: MPI+OPENMP: BODIES-SPEED RELATIONSHIP

4 CONCLUSION

1. In the MPI version, we see in small and medium number of bodies (figure size < 300), the speed will first increase then decrease as the core number increases. While in the cases of large figure size (figure size >= 300), the speed will keep increasing as the number of cores increases. The probable reason might be that for the first case, the costs of create processes and passing messages are greater than the benefit of parallel computing, because the problem size is not large. In the second case, the costs of create processes and passing messages are smaller than the benefit of parallel computing, because the problem size is large. According to the Gustafson's Law, the increasing of cores can speed up the computation with increasing problem size. In first case, we can also observe that 32, 64, 96, 128 are turning points for some configurations. It is because the cross node communication occurs, which is time-consuming.

2. In the Pthread version, we can observe that when the number of thread increases, the speed decreases. The number of thread has larger effect on small figure size than large figure size. The probable reason is that, on the one hand, the task of Mandelbrot set computing is not an IO bound task. In other words, it requires a lot of computation, but within the part that is profiled, it does not have IO operations. Thus, multi-threads programming brings little benefits. On the other hand, the creation and scheduling of threads requires extra time. And as threads are executed concurrently within a core, there might be conflicts among threads. So multi-thread programming costs more time. And we allocate threads within one node. A node has maximum 32 cores. So when the number of thread is greater than 32, the operating system will schedule the threads. Some threads will be suspended. This causes the overhead.

3. In the OpenMP case, as it uses pthread, the performance is similar to the pthread version. And thats also why pthread performs better than OpenMP in terms of speed.

4. In the CUDA version, the speed increases as the number of threads increases, the speed increases. The probable reason is that the memory is shared among the threads, which little cost to transfer the messages. In one block of the GPU, we can launch 1024 threads simultaneously. The cost is small, but the program can benefit from the parallelism.

5. In the MPI+OpenMP version, when node == 1, there is only 32 cores, so the maximum number of threads that are run simultaneously is 32. The operating system is responsible for scheduling the threads. When n == 2, there are 64 cores. In this case, there are still some threads will be suspended. But there exists cross nodes communication, the cost is greater than the benefit of more cores. When n == 3, there are 96 cores, so each thread can run on one core, there is no need to suspend a thread. So the speed is faster than the

case where n == 2. As there still exists cross node communication, the cost is large. The speed is slower than the case where n == 1.

6. The sequential version and the MPI version performs good when the number of bodies is large. When the number of bodies is small, we cannot gain much from parallelism. The cost of forking a process is higher than forking a thread, and parallelism.
7. CUDA version performs best when the number of thread or the number of process is large. The probable reason might be that the structure of GPU makes creating a thread more efficient.

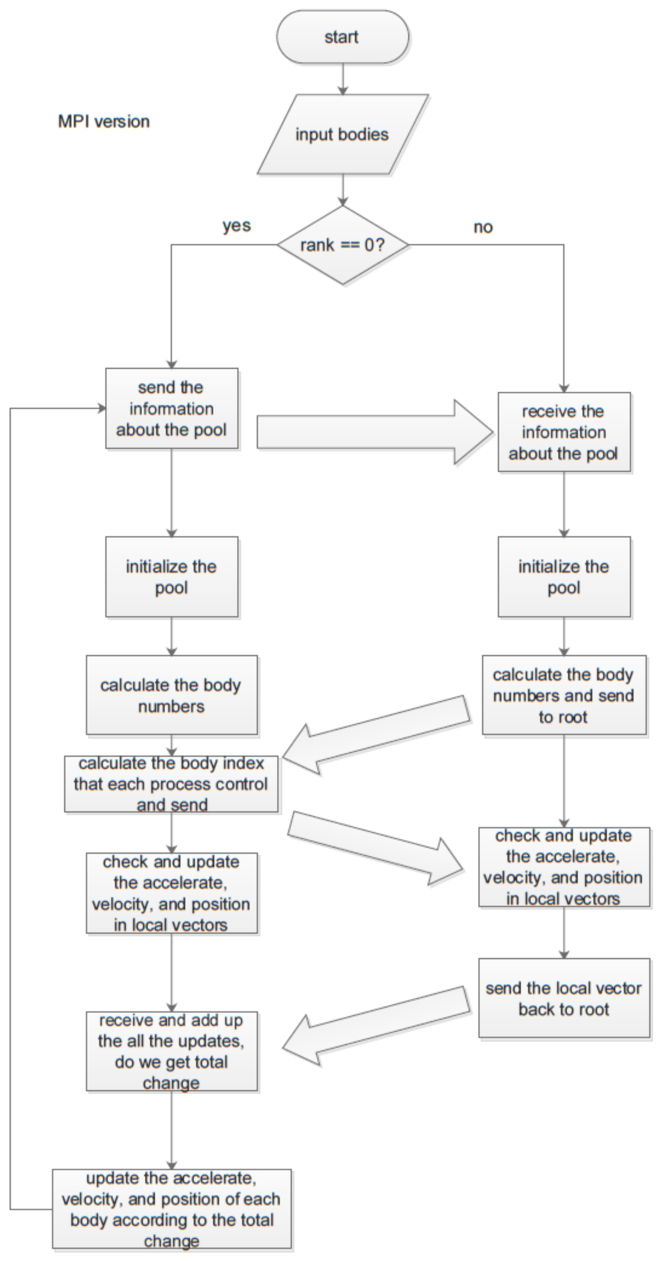

MPI version
start
input bodies
rank == 0?
yes
no
send the information about the pool
receive the information about the pool
initialize the pool
initialize the pool
calculate the body numbers
calculate the body numbers and send to root
calculate the body index that each process control and send
check and update the accelerate, velocity, and position in local vectors
check and update the accelerate, velocity, and position in local vectors
send the local vector back to root
receive and add up the all the updates, do we get total change
update the accelerate, velocity, and position of each body according to the total change

FIGURE 22: FLOWCHART OF THE MPI VERSION

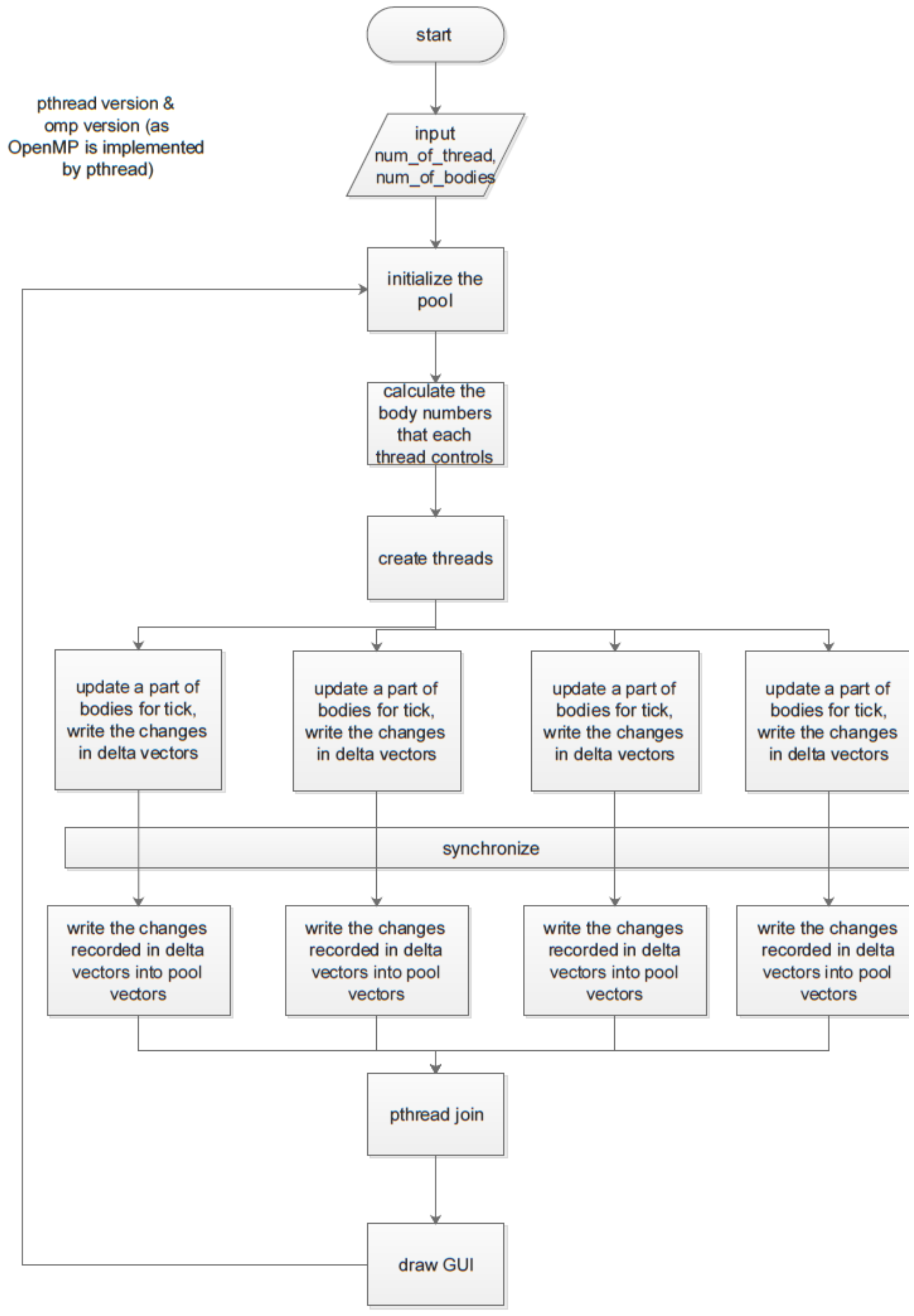

pthread version & omp version (as OpenMP is implemented by pthread)
start
input num_of_thread, num_of_bodies
initialize the pool
calculate the body numbers that each thread controls
create threads
update a part of bodies for tick, write the changes in delta vectors
update a part of bodies for tick, write the changes in delta vectors
update a part of bodies for tick, write the changes in delta vectors
update a part of bodies for tick, write the changes in delta vectors
synchronize
write the changes recorded in delta vectors into pool vectors
write the changes recorded in delta vectors into pool vectors
write the changes recorded in delta vectors into pool vectors
write the changes recorded in delta vectors into pool vectors
pthread join
draw GUI

FIGURE 23: FLOWCHART OF THE PTHREAD VERSION AND THE OPENMP VERSION

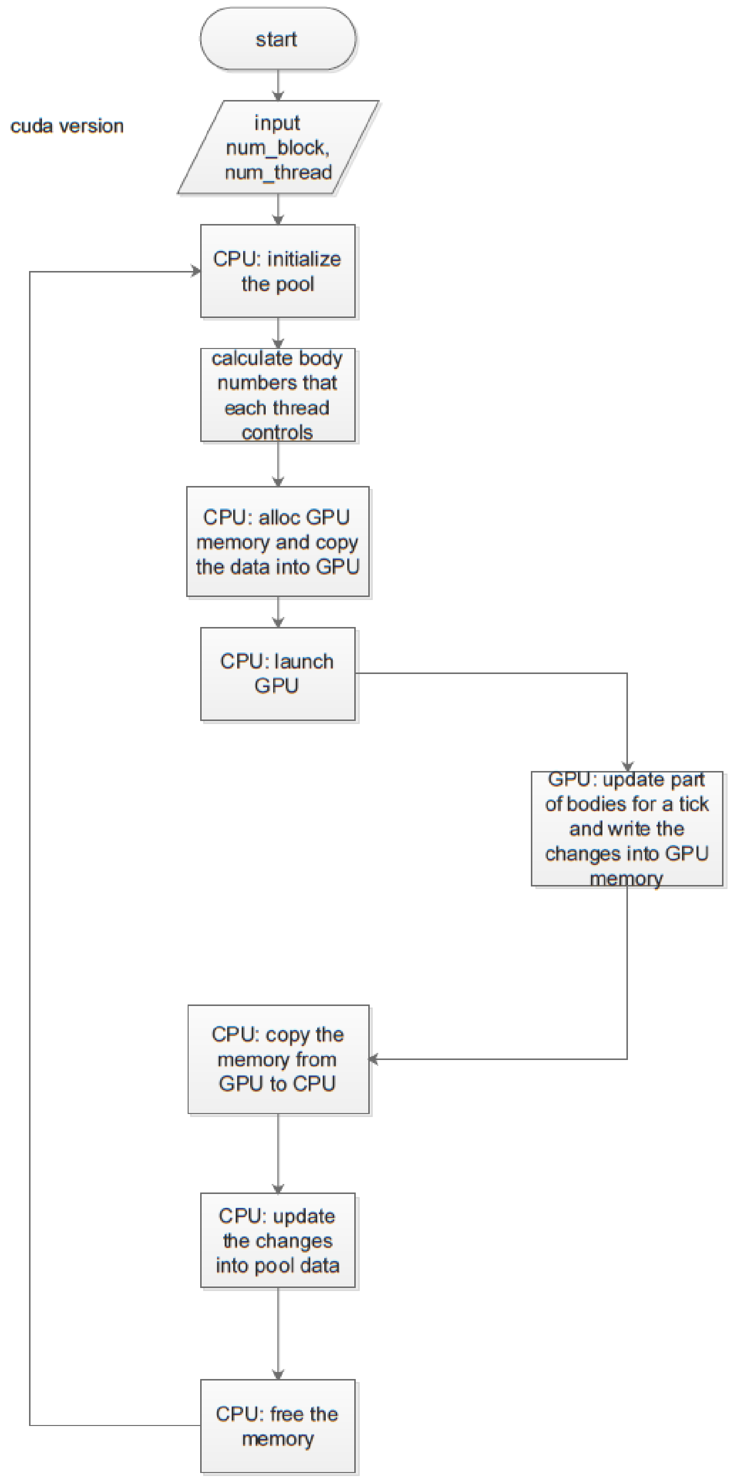

start
cuda version
input num_block, num_thread
CPU: initialize the pool
calculate body numbers that each thread controls
CPU: alloc GPU memory and copy the data into GPU
CPU: launch GPU
GPU: update part of bodies for a tick and write the changes into GPU memory
CPU: copy the memory from GPU to CPU
CPU: update the changes into pool data
CPU: free the memory

FIGURE 24: FLOWCHART OF THE CUDA VERSION

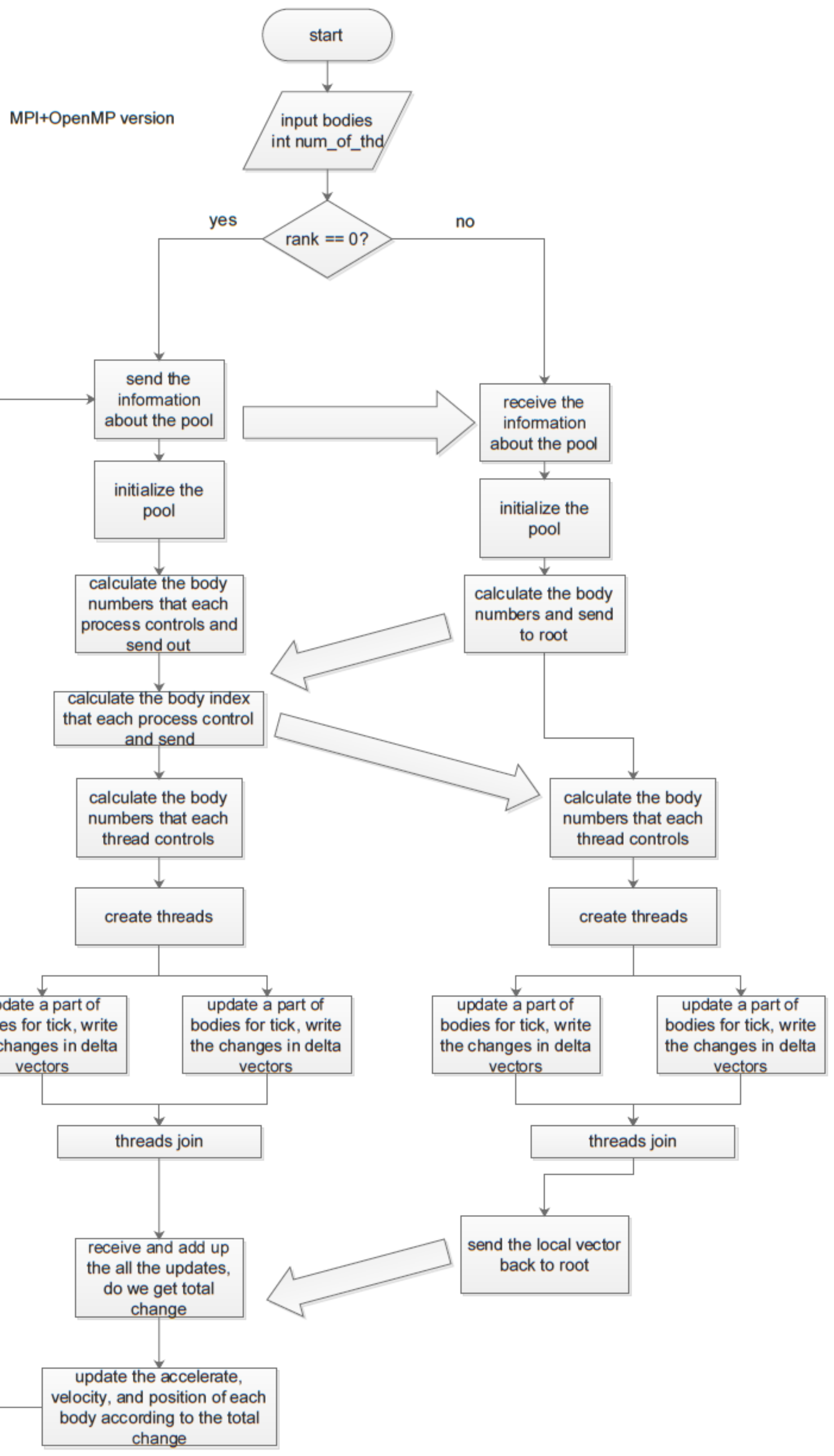

start
MPI+OpenMP version
input bodies
int num_of_thd
rank == 0?
yes
no
send the information about the pool
receive the information about the pool
initialize the pool
initialize the pool
calculate the body numbers that each process controls and send out
calculate the body numbers and send to root
calculate the body index that each process control and send
calculate the body numbers that each thread controls
calculate the body numbers that each thread controls
create threads
create threads
update a part of bodies for tick, write the changes in delta vectors
update a part of bodies for tick, write the changes in delta vectors
update a part of bodies for tick, write the changes in delta vectors
update a part of bodies for tick, write the changes in delta vectors
threads join
threads join
send the local vector back to root
receive and add up the all the updates, do we get total change
update the accelerate, velocity, and position of each body according to the total change

FIGURE 25: FLOWCHART OF THE HYBRID VERSION