The Chinese University of Hong Kong,
Shenzhen

CSC4005 Parallel Programming

Homework 3 Report

Name: Xiang Fei
Student ID: 120090414
Email: xiangfei@link.cuhk.edu.cn
Date: 2022.11

Table of Contents

List of Figures

List of Tables

1 Introduction

Assignment 3 requires us to design a program for N-body simulation and render the points on screen with a type of GUI. In a two-dimension astronomical space, several bodies move under the universal gravitation. The physical formula is expressed as:

$$F = \frac{Gm_1m_2}{r^2} \tag{1}$$

With the force, the bodies obtain some acceleration and gain velocity changes and then positional changes. The bodies are initially at rest, whose positions and masses are randomly generated. There is a bound for the body's movement which prevents the bodies from moving outward. Besides the gravitational force, the collision between balls and balls or balls and walls should be taken into account. Our task in the assignment is to keep computing the movement of the bodies in a certain elapse and rendering all the planets in the screen using the GUI given by the template and explore the relationship between computation speed and different parameters.

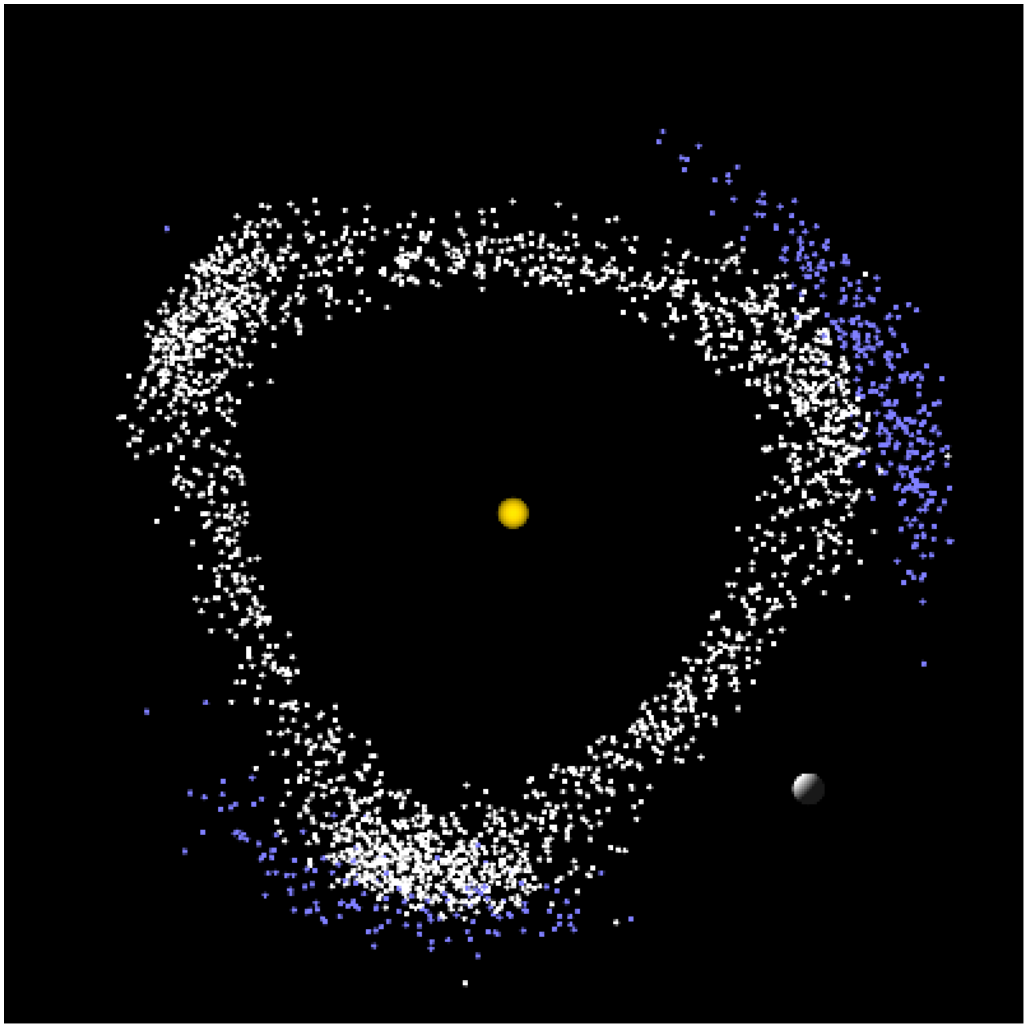


Figure 1: N-body simulation

In this homework, we need to implement the sequential version code, parallel versions using MPI and Pthread and openMP and CUDA approaches. The combination of MPI & openMP version serves as the bonus.

MPI (Massage Passing Interface) is the standard of message passing function library that includes many interfaces. It is a parallel technology based on the communication between processes. A sample MPI interface is shown as follows. MPI_Send is one of the most popular functions in MPI.

Pthread refers to POSIX threads, which is an execution model that exists independently from a language and a parallel execution model. It is primitive and practical in C programming. Here shows the basic grammar of creating a thread and synchronizing the thread.

OpenMP is an application programming interface that supports multi-platform shared-memory multiprocessing programming. Only certain #pragma is required to add before the codes to realize parallel programming. No extra initialization is needed. It is probably one of the most convenient approaches among the four.

CUDA refers to Compute Unified Device Architecture. It is a parallel computing platform and application interface that utilizes the power of GPU. Only NVIDIA GPU cards are supported. The CUDA approach includes many complex concepts including signs (device, host, global), grid and block, memory management, and so on.

Although the parallel computing approaches may be different, the core part of this assignment is the same, which is to partition the body calculation reasonably and allocate to different processes/threads. Proper synchronization and data aggregation strategies also require careful consideration.

2 Design and Method

In this part, I illustrate the design logic of each version of program. In addition, some details of the code implementation will also be elaborated. All of the implementations are based on the template given by TAs.

2.1 sequential version

In the implementation of sequential version program, the core is two functions: **update_position** and **update_velocity**, which realize the logic of changing the position of each particle and the velocity of each particle respectively. And the logic of ball-to-ball collision and ball-to-wall collision is also implemented within this two functions. Therefore, I will introduce these two important functions (also the basic of the parallel implementations).

2.1.1 update_position

```
void update_position(double *x, double *y, double *vx, double *vy, int n) {
    //TODO: update position
    for(int i=0;i<n;i++){
        x[i] = x[i] + vx[i]*dt;
        y[i] = y[i] + vy[i]*dt;
        // check the ball & wall collision
        if(x[i]<0){
            vx[i] = -vx[i];
            x[i] = -x[i];
        }
        else if(x[i]>bound_x){
            vx[i] = -vx[i];
            x[i] = 2*bound_x-x[i];
        }
        if(y[i]<0){
            vy[i] = -vy[i];
            y[i] = -y[i];
        }
        else if(y[i]>bound_y){
            vy[i] = -vy[i];
            y[i] = 2*bound_y-y[i];
        }
    }
}
```

Figure 2: The code of sequential update_function

The inputs of this function is: an array which stores the x position of each particle, an array which stores the y position of each particle, an array which stores the x velocity of each particle, an array which stores the y velocity of each particle and an integer which is the total number of the particles.

In this function, I first change the position of each particle by the following formula:

$$x = x + v_x * dt \tag{2}$$

$$y = y + v_y * dt \tag{3}$$

And then, I check the ball-to-wall collision in this function. I consider in two dimensions, for both x and y, if the particle is out of the bound, then its velocity in this dimension will be inverse and the position will be symmetric with the outer point corresponding to the bound line.

2.1.2 update_velocity

```
void update_velocity(double *m, double *x, double *y, double *vx, double *vy, int n) {
    //TODO: calculate force and acceleration, update velocity
    for(int i=0;i<n;i++){
        double axi = 0;
        double ayi = 0;
        for(int j=0;j<n;j++){
            if(j==i){
                continue;
            }
            double distance2 = pow(x[i]-x[j],2)+pow(y[i]-y[j],2);
            if(distance2<=4*radius2){ // ball i and j have a collision
                if(vx[i]*vx[j]<0 || (vx[i]>=0&&vx[j]>=0&&vx[i]>vx[j]) || (vx[i]<=0&&vx[j]<=0&&vx[i]>vx[j])){
                    vx[i] = ((m[i]-m[j])*vx[i]+2*m[j]*vx[j])/(m[i]+m[j]);
                    vx[j] = ((m[j]-m[i])*vx[i]+2*m[i]*vx[i])/(m[i]+m[j]);
                }
                if(vy[i]*vy[j]<0 || (vy[i]>=0&&vy[j]>=0&&vy[i]>vy[j]) || (vy[i]<=0&&vy[j]<=0&&vy[i]>vy[j])){
                    vy[i] = ((m[i]-m[j])*vy[i]+2*m[j]*vy[j])/(m[i]+m[j]);
                    vy[j] = ((m[j]-m[i])*vy[i]+2*m[i]*vy[i])/(m[i]+m[j]);
                }
                axi = 0;
                ayi = 0;
                break;
            }
            else{
                double forcex = gravity_const*m[i]*m[j]*(x[j]-x[i])/(pow(distance2,1.5)+err);
                double forcey = gravity_const*m[i]*m[j]*(y[j]-y[i])/(pow(distance2,1.5)+err);
                axi += forcex / m[i];
                ayi += forcey / m[i];
            }
        }
        vx[i] = vx[i] + axi*dt;
        vy[i] = vy[i] + ayi*dt;
    }
}
```

Figure 3: The code of sequential update_velocity

In this function, I consider the ball-to-ball collision and also the force from other particles.

There are two loops (outer loop and inner loop) in this function. The outer index i is the particle we focus on in this outer iteration. For this particle, we consider the interaction between it and all other particles (represent by j in each inner iteration). First, we check whether there is a collision between i and j. The condition is:

1 . The distance between particle i and j is small then or equal to the diameter of the particle (so the square of distance is small then or equal to 4 times the radius's square)

2 . The two particles moving in opposite directions or two particles moving in the same direction but the lagging particle is faster.

The collision is modeled by using the law of conservation of momentum, and assumes elastic collisions (i.e. collision restitution (e) equal to 1):

$$m_i v_{i0} + m_j v_{j0} = m_i v_i + m_j v_j \tag{4}$$

$$e = \frac{v_j - v_i}{v_{i0} - v_{j0}} = 1 \tag{5}$$

Therefore, we have:

$$v_i = \frac{(m_i - m_j)v_{i0} + 2m_j v_{j0}}{m_i + m_j} \tag{6}$$

$$v_j = \frac{(m_j - m_i)v_{j0} + 2m_i v_{i0}}{m_i + m_j} \tag{7}$$

For each direction (x and y since freedom degree is 2), the above formula is satisfied.

I also consider that, when there is a collision happens, then the velocity of these two particles will not be affected by the force between these particles and from other particles. This is because the force cased by collision is vary large (can be analyzed by the impulse theorem):

$$F\Delta t = m\Delta v \tag{8}$$

And here, t is very small, so F in fact is vary large, we ignore the influence from other forces.

If there is no collision for the considered particle, then we compute the accelerate by calculating the force from all other particles for each direction:

$$F_x = \frac{Gm_i m_j}{r^2} \times \frac{x_j - x_i}{r} \tag{9}$$

$$F_y = \frac{Gm_i m_j}{r^2} \times \frac{y_j - y_i}{r} \tag{10}$$

$$a_x = \frac{F_x}{m_i} \tag{11}$$

$$a_y = \frac{F_y}{m_i} \tag{12}$$

Then, the velocity will change as the following formula:

$$v_x = v_x + a_x d_t \tag{13}$$

$$v_y = v_y + a_y d_t \tag{14}$$

Last, in the master function, we invoke **update_velocity** and **update_position** in each iteration.

```
for (int i = 0; i < n_iteration; i++){
    std::chrono::high_resolution_clock::time_point t1 = std::chrono::high_resolution_clock::now();

    update_velocity(m, x, y, vx, vy, n_body);
    update_position(x, y, vx, vy, n_body);


    std::chrono::high_resolution_clock::time_point t2 = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> time_span = t2 - t1;
```

Figure 4: The code of sequential master (part)

The update algorithm is like the following graph:

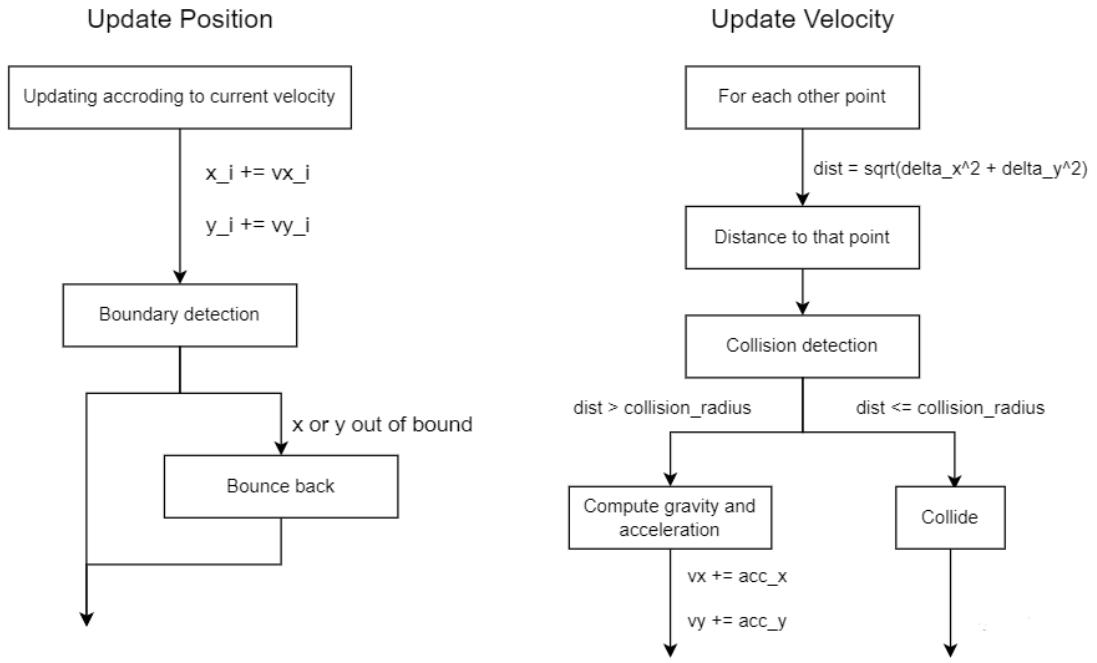


Figure 5: The update algorithm

2.2 pthread version

For the pthread version, the first thing is to make sure that how to parallel, or we can say, how to partition the problem. In my program, for both the velocity computation and position computation, I use a dynamic scheduling. At the beginning, each thread compute for one particle, and when one of the threads finish its computing, then the thread will compute the next particle. We can say "first finish, first work to another". When all the particles are computed, then we finished. The following graph shows the scheduling method of my pthread program (I take three threads as an example).

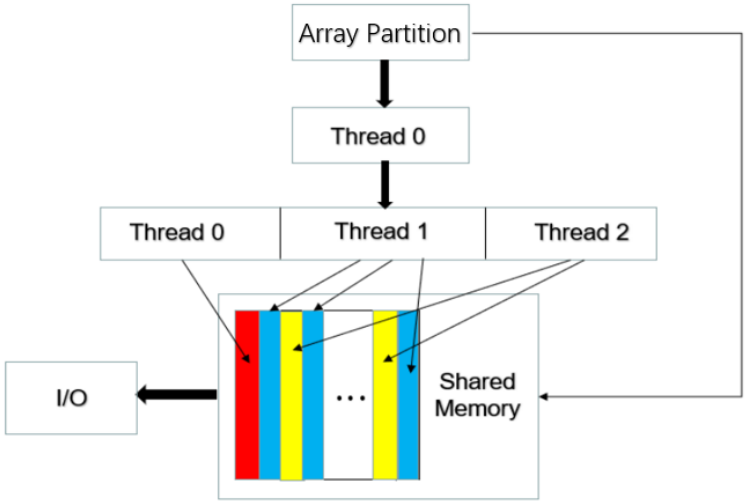


Figure 6: The dynamic scheduling of pthread

The array in the graph is the information of particles that should be computed, including the position (x and y) and the velocity (vx and vy). You can see the graph, at the beginning, thread 0, 1, 2 are assigned to the first three particles, then the thread 1 finishs it work, so this thread can go to work for the fourth particle, and so on, until all the particles are handled.

Then, I will introduce the code in detail.

First is the global variables I added.

```
pthread_mutex_t mutex;
pthread_barrier_t mybarrier;
int n_body;
int n_iteration;
int count1=0;
int count2=0;
```

Figure 7: The global variables of pthread

mutex is used to prevent conflicts. **mybarrier** is used to implement synchronization. **count1** and **count2** are used to implement dynamic scheduling, the details will be introduced in the later part, which is the introduction of important implementations (I only focus on the difference part with the sequential version, so do all the following part).

2.2.1 update_position

```
void update_position(double *x, double *y, double *vx, double *vy, int n, int tid, int num_thd) {
    //TODO: update position
    count1 = num_thd-1;
    int current = tid;
    while(count1<n && current<n){
        x[current] = x[current] + vx[current]*dt;
        y[current] = y[current] + vy[current]*dt;
        // check the ball & wall collision
        if(x[current]<0){
            vx[current] = -vx[current];
            x[current] = -x[current];
        }
        else if(x[current]>bound_x){
            vx[current] = -vx[current];
            x[current] = 2*bound_x-x[current];
        }
        if(y[current]<0){
            vy[current] = -vy[current];
            y[current] = -y[current];
        }
        else if(y[current]>bound_y){
            vy[current] = -vy[current];
            y[current] = 2*bound_y-y[current];
        }
        pthread_mutex_lock(&mutex);
        current = count1 + 1;
        count1++;
        pthread_mutex_unlock(&mutex);
        if(count1>=n) break;
    }
}
```

Figure 8: The update_position function of pthread

In this function, **count1** is used to find the next free (has not been handled) particle. And **current** is the position of the current handling particle. Here, **mutex** is used to avoid race, and to make sure that the next particle will be handled by only one thread. In fact, function **update_velocity** uses the same method.

2.2.2 Args

```
typedef struct {
    //TODO: specify your arguments for threads
    double *m;
    double *x;
    double *y;
    double *vx;
    double *vy;
    int n;
    int tid;
    int num_thd;
    //TODO END
} Args;
```

Figure 9: The Args structure of pthread

This structure stores the arguments for threads. **m** is the mass array, **x** is the x position array, **y** is the y position array, **vx** is the x velocity array, **vy** is the y velocity array, **n** is the number of particles, **tid** is the thread id, **num_thd** is the number of threads.

2.2.3 worker

```
void* worker(void* args) {
    //TODO: procedure in each threads
    Args* my_arg = (Args*) args;
    double *m = my_arg->m;
    double *x = my_arg->x;
    double *y = my_arg->y;
    double *vx = my_arg->vx;
    double *vy = my_arg->vy;
    int n = my_arg->n;
    int tid = my_arg->tid;
    int num_thd = my_arg->num_thd;

    update_velocity(m, x, y, vx, vy, n, tid, num_thd);

    pthread_barrier_wait(&mybarrier);

    update_position(x, y, vx, vy, n, tid, num_thd);
    // TODO END
}
```

Figure 10: The worker function of pthread

In this function, the most importance thing is the use of **pthread_barrier_wait**, which is used to implement synchronization. By using the synchronization technique, we can make sure that we will update the positions of particles after we finish the computing of their velocity.

Finally, we can get the flow chart of my pthread program.

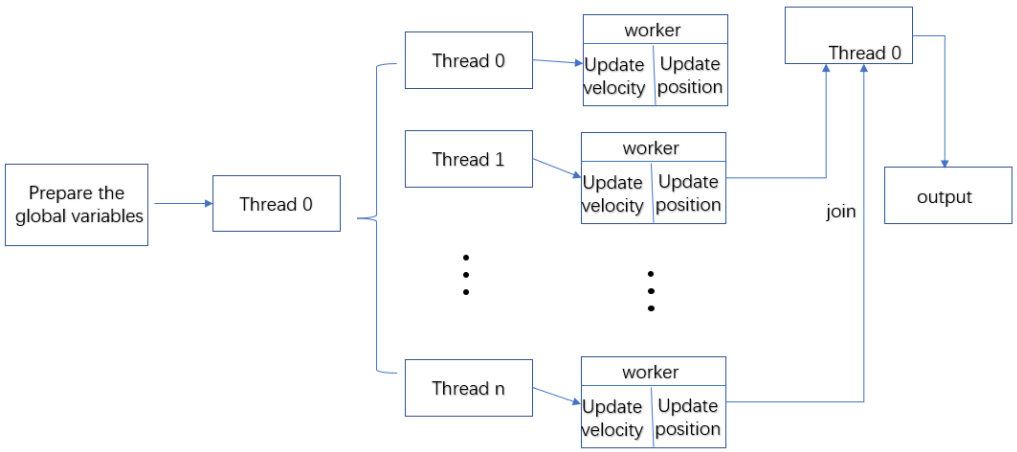


Figure 11: The flow chart of pthread

Also, we can have the coding logic graph of pthread:

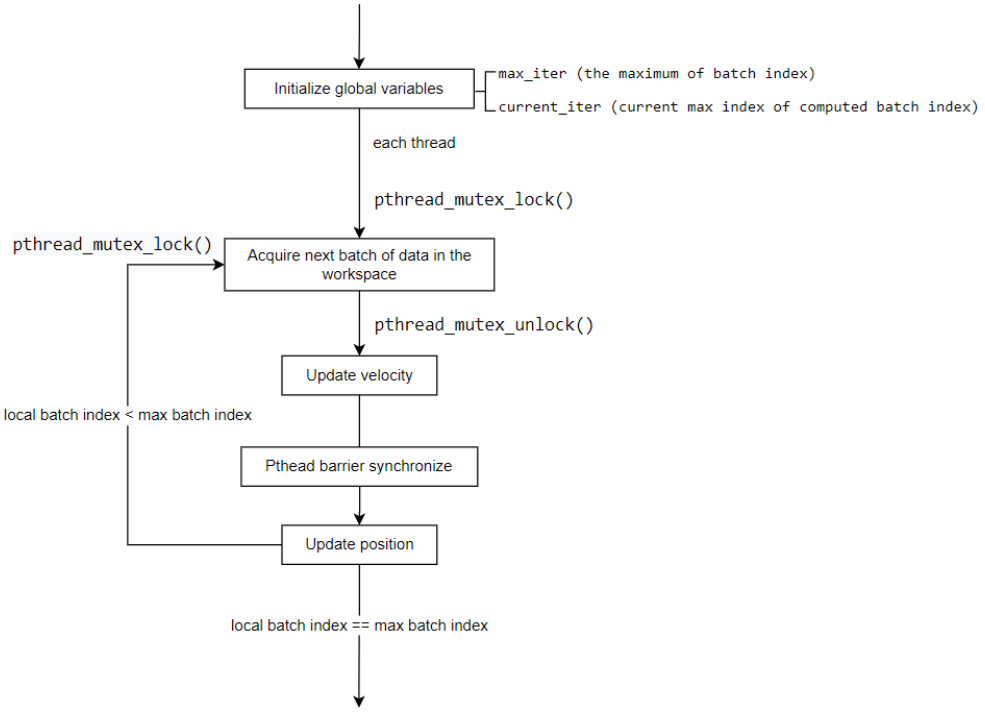


Figure 12: The coding logic of pthread

2.3 MPI version

In the MPI version, I just use a static scheduling, which means I assigned an equal amount of particles to each thread to calculate. If there is remainder, then I just let the master process to compute the remaining particles.

The implementation of my MPI version program is based on **MPI_Bcast** and **MPI_Scatter**.

```
if(my_rank==0){
    generate_data(total_m, total_x, total_y, total_vx, total_vy, n_body);
}
MPI_Barrier(MPI_COMM_WORLD);
MPI_Bcast(total_m, n_body, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Scatter(total_m, my_length, MPI_DOUBLE, local_m, my_length, MPI_DOUBLE, 0, MPI_COMM_WORLD);

for (int i = 0; i < n_iteration; i++){
    if(my_rank==0){
        t1 = std::chrono::high_resolution_clock::now();
    }
    // TODO: MPI routine

    MPI_Bcast(total_x, n_body, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(total_y, n_body, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(total_vx, n_body, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(total_vy, n_body, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Scatter(total_x, my_length, MPI_DOUBLE, local_x, my_length, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Scatter(total_y, my_length, MPI_DOUBLE, local_y, my_length, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Scatter(total_vx, my_length, MPI_DOUBLE, local_vx, my_length, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Scatter(total_vy, my_length, MPI_DOUBLE, local_vy, my_length, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Figure 13: The use of MPI_Bcast and MPI_Scatter of MPI

Here, **MPI_Bcast** is used to broadcast the total mass, x, y, vx, vy array to each process, since when we consider the force between particles in each process, we need to know the information of all particles. And **MPI_Scatter** is used to distribute the work to each process. By using this function, each process gets the information array it should handle. In addition, **MPI_Barrier** here is used to synchronize, so that the broadcast will be executed after the master process has finished the data generation step.

```
if(my_rank==0){
    generate_data(total_m, total_x, total_y, total_vx, total_vy, n_body);
}
MPI_Barrier(MPI_COMM_WORLD);
MPI_Bcast(total_m, n_body, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Scatter(total_m, my_length, MPI_DOUBLE, local_m, my_length, MPI_DOUBLE, 0, MPI_COMM_WORLD);

for (int i = 0; i < n_iteration; i++){
    if(my_rank==0){
        t1 = std::chrono::high_resolution_clock::now();
    }
    // TODO: MPI routine

    MPI_Bcast(total_x, n_body, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(total_y, n_body, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(total_vx, n_body, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(total_vy, n_body, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Scatter(total_x, my_length, MPI_DOUBLE, local_x, my_length, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Scatter(total_y, my_length, MPI_DOUBLE, local_y, my_length, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Scatter(total_vx, my_length, MPI_DOUBLE, local_vx, my_length, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Scatter(total_vy, my_length, MPI_DOUBLE, local_vy, my_length, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Figure 14: The use of MPI_Bcast and MPI_Scatter of MPI

And after each process finish its work, we need to use **MPI_Gather** to gather the updated data, so that we can give a right output.

```
MPI_Gather(local_x, my_length, MPI_DOUBLE, total_x, my_length, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Gather(local_y, my_length, MPI_DOUBLE, total_y, my_length, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Gather(local_vx, my_length, MPI_DOUBLE, total_vx, my_length, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Gather(local_vy, my_length, MPI_DOUBLE, total_vy, my_length, MPI_DOUBLE, 0, MPI_COMM_WORLD);
// TODO End
```

Figure 15: The use of MPI_Gather of MPI

Finally, we can also get the flow chart of my MPI version program.

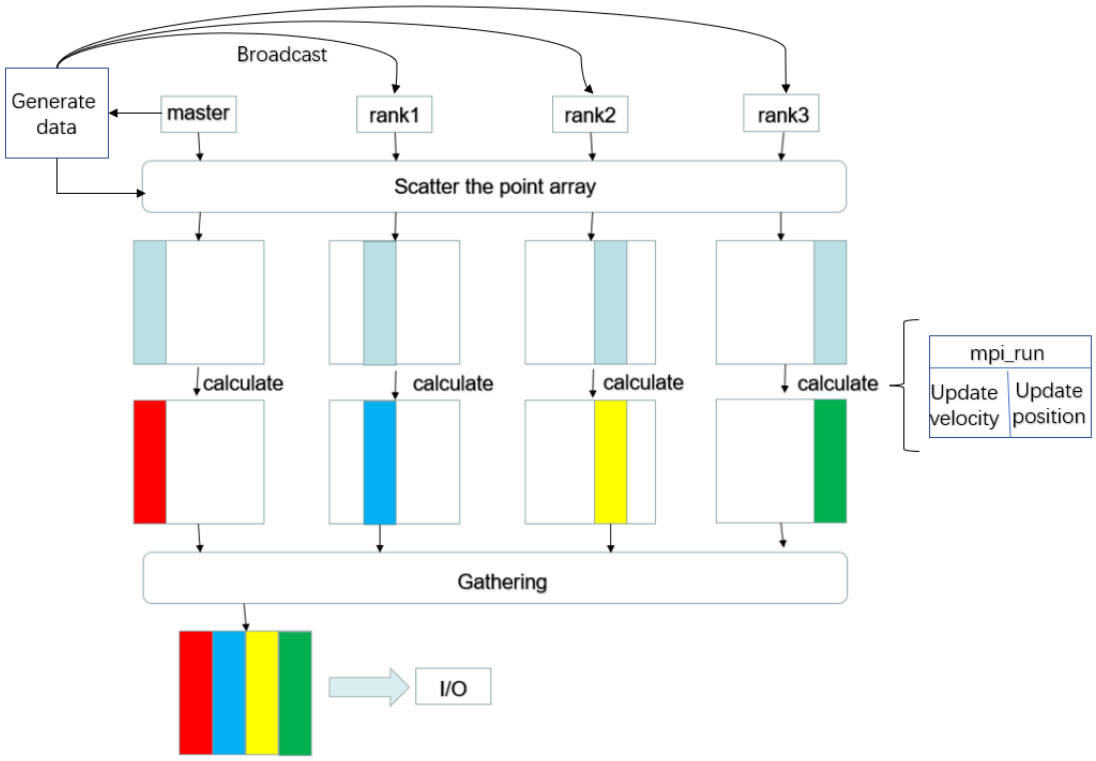


Figure 16: The flow chart of MPI

2.4 openmp version

For the openmp version implementation, we just need to make a small change with the sequential version. For both **update_position** and **update_velocity** functions, we just remove the outer for loop (the loop will be done in **master** function to implement parallelization), and we need to add the index input.

```
void update_position(double *x, double *y, double *vx, double *vy, int i) {
    //TODO: update position
    x[i] = x[i] + vx[i]*dt;
    y[i] = y[i] + vy[i]*dt;
    // check the ball & wall collision
    if(x[i]<0){
        vx[i] = -vx[i];
        x[i] = -x[i];
    }
    else if(x[i]>bound_x){
        vx[i] = -vx[i];
        x[i] = 2*bound_x-x[i];
    }
    if(y[i]<0){
        vy[i] = -vy[i];
        y[i] = -y[i];
    }
    else if(y[i]>bound_y){
        vy[i] = -vy[i];
        y[i] = 2*bound_y-y[i];
    }
}
```

Figure 17: The update_position function of openmp version

```
void update_velocity(double *m, double *x, double *y, double *vx, double *vy, int i, int n, double axi, double ayi, int j) {
    //TODO: calculate force and acceleration, update velocity
    for(int j=0;j<n;j++){
        if(j==i){
            continue;
        }
        double distance2 = pow(x[i]-x[j],2)+pow(y[i]-y[j],2);
        if(distance2<=4*radius2){ // ball i and j have a collision
            if(vx[i]*vx[j]<0 || (vx[i]>=0&&vx[j]>=0&&vx[i]>vx[j]) || (vx[i]<=0&&vx[j]<=0&&vx[i]>vx[j])){
                vx[i] = ((m[i]-m[j])*vx[i]+2*m[j]*vx[j])/(m[i]+m[j]);
                vx[j] = ((m[j]-m[i])*vx[i]+2*m[i]*vx[i])/(m[i]+m[j]);
            }
            if(vy[i]*vy[j]<0 || (vy[i]>=0&&vy[j]>=0&&vy[i]>vy[j]) || (vy[i]<=0&&vy[j]<=0&&vy[i]>vy[j])){
                vy[i] = ((m[i]-m[j])*vy[i]+2*m[j]*vy[j])/(m[i]+m[j]);
                vy[j] = ((m[j]-m[i])*vy[i]+2*m[i]*vy[i])/(m[i]+m[j]);
            }
            axi = 0;
            ayi = 0;
            break;
        }
        else{
            double forcex = gravity_const*m[i]*m[j]*(x[j]-x[i])/(pow(distance2,1.5)+err);
            double forcey = gravity_const*m[i]*m[j]*(y[j]-y[i])/(pow(distance2,1.5)+err);
            axi += forcex / m[i];
            ayi += forcey / m[i];
        }
    }
    vx[i] = vx[i] + axi*dt;
    vy[i] = vy[i] + ayi*dt;
}
```

Figure 18: The update_velocity function of openmp version

And then, I use #pragma in the **master** function before the for loop, so that we can use multi-threads to solve the problem and implement parallelization.

```
for (int i = 0; i < n_iteration; i++){
    double axi;
    double ayi;
    int j;
    std::chrono::high_resolution_clock::time_point t1 = std::chrono::high_resolution_clock::now();

    //TODO: choose better threads configuration
    omp_set_num_threads(n_omp_threads);
    #pragma omp parallel for private (axi,ayi,j)
    for (int i = 0; i < n_body; i++) {
        update_velocity(m, x, y, vx, vy, i, n_body, axi, ayi, j);
    }

    omp_set_num_threads(n_omp_threads);
    #pragma omp parallel for
    for (int i = 0; i < n_body; i++) {
        update_position(x, y, vx, vy, i);
    }

    std::chrono::high_resolution_clock::time_point t2 = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> time_span = t2 - t1;
```

Figure 19: The parallelization method of openmp version

please pay attention that here we should make the variables *axi*, *ayi* and *j* private.

Finally, we can get the flow chart of my openmp version program.

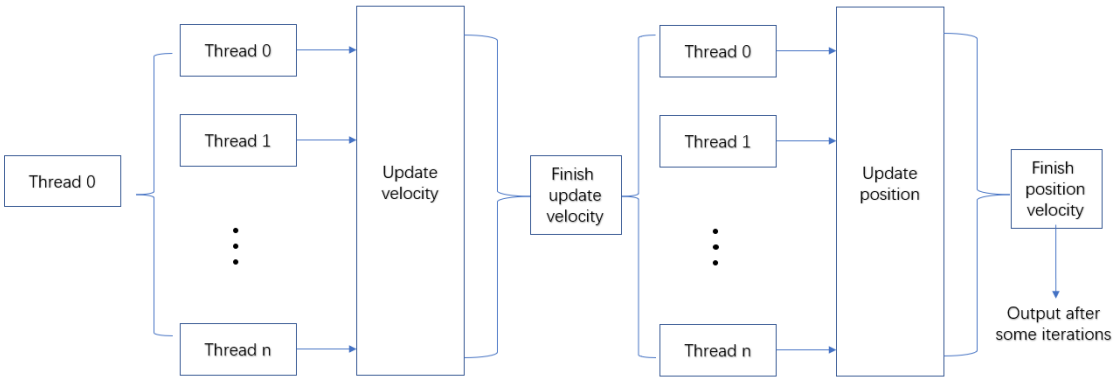


Figure 20: The flow chart of openmp version

2.5 cuda version

For my cuda version implementation, I also remove the outer for loop, since I can let each thread finishes one outer for loop work. and in order to maintain the data consistency, I use "__syncthreads()" function after the computation of velocity for each thread, therefore, the position will change only when all the particles' velocity has changed.

In addition, I use **cudaMalloc** to allocate the memory in device, and I use **cudaMemcpy** to implement the memory copy from host to device or from device to host.

```
cudaMalloc(&device_m, n_body * sizeof(double));
cudaMalloc(&device_x, n_body * sizeof(double));
cudaMalloc(&device_y, n_body * sizeof(double));
cudaMalloc(&device_vx, n_body * sizeof(double));
cudaMalloc(&device_vy, n_body * sizeof(double));

cudaMemcpy(device_m, m, n_body * sizeof(double), cudaMemcpyHostToDevice);
cudaMemcpy(device_x, x, n_body * sizeof(double), cudaMemcpyHostToDevice);
cudaMemcpy(device_y, y, n_body * sizeof(double), cudaMemcpyHostToDevice);
cudaMemcpy(device_vx, vx, n_body * sizeof(double), cudaMemcpyHostToDevice);
cudaMemcpy(device_vy, vy, n_body * sizeof(double), cudaMemcpyHostToDevice);

int n_block = n_body / block_size + 1;

for (int i = 0; i < n_iteration; i++){
    std::chrono::high_resolution_clock::time_point t1 = std::chrono::high_resolution_clock::now();

    update_velocity<<<n_block, block_size>>>(device_m, device_x, device_y, device_vx, device_vy, n_body);
    update_position<<<n_block, block_size>>>(device_x, device_y, device_vx, device_vy, n_body);

    cudaMemcpy(x, device_x, n_body * sizeof(double), cudaMemcpyDeviceToHost);
    cudaMemcpy(y, device_y, n_body * sizeof(double), cudaMemcpyDeviceToHost);

    std::chrono::high_resolution_clock::time_point t2 = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> time_span = t2 - t1;
```

Figure 21: The use of cudaMalloc and cudaMemcpy of cuda version

Finally, the flow chart of my cuda version program can be obtained:

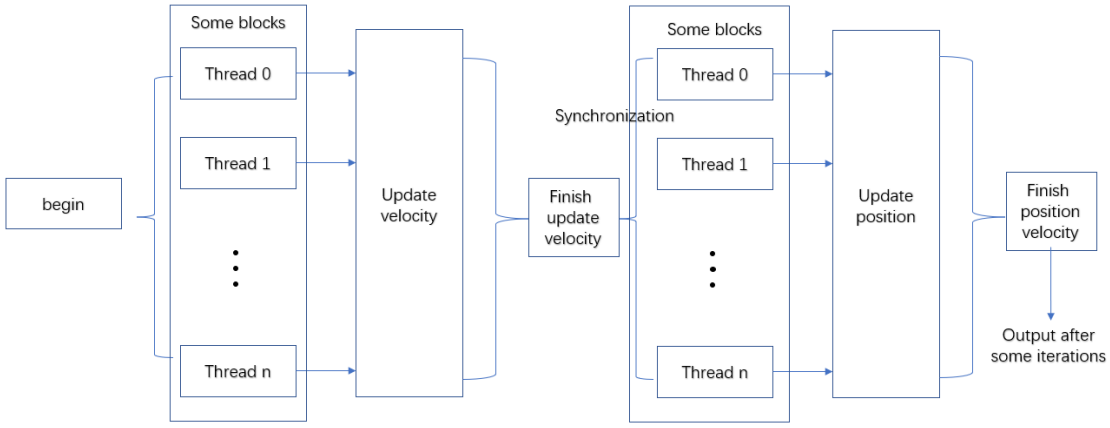


Figure 22: The flow chart of cuda version

2.6 bonus

For my bonus work, I use both MPI and openmp the parallize the outer for loop, which means I can distribute the particle computation work to different threads of different process. In fact, we just need to add #pragma before the outer for loop in both **update_velocity** and **update_position** functions, and the remaining is the same with the MPI version (also use $\mathbf{MPI}_B cast$,

```
void update_velocity(double *local_vx, double *local_v
    //TODO: calculate force and acceleration, update v
    double axi;
    double ayi;
    int j;
    omp_set_num_threads(n_omp_threads);
    #pragma omp parallel for private (axi,ayi,j)
    for(int i=0;i<local_n;i++){
        axi = 0;
        ayi = 0;
        for(j=0;j<n;j++){
            if(j==i+my_rank*local_n){
```

Figure 23: The modification (compare with MPI version) in update_velocity function to implement openmp in the bonus version

Finally, we can get the flow chart of my bonus version program.

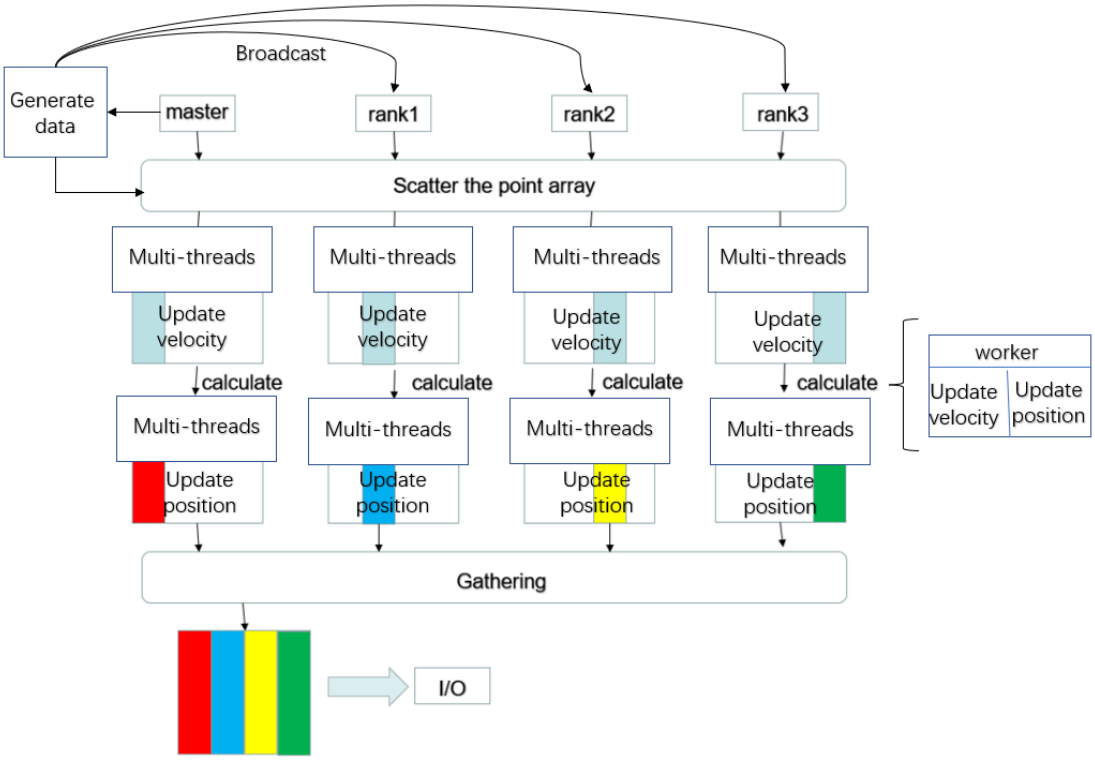


Figure 24: The flow chart of my bonus version implementation

3 Execution

My code executes on the remote cluster and my virtual machine (for GUI). The cluster resource limits is showed in the following graph.

Cluster Resource Limit

Oct 1, 2022

partition	Debug	Project
`MaxNodes` (max number of **nodes** allocated to a job)	1	1
`MaxCPUsPerNode` (max number of **cpus on a node** allocated to a job)	8	40
Max number of **total cpu cores** allocated to a job	1*8=8	1*40=40
`MaxTime` (max **running time** of a job)	60min	10min
`MaxJobsPerUser` (max number of **running** jobs of a user at a given time)	2	1
`MaxSubmitJobsPerUser` (max number of jobs **submitted** of a user at a given time)	10	100
Total number of nodes in this partition (we will add more if not enough)	9	18

For time consuming jobs with a few cores, you can use `Debug` partition.

For jobs requiring many cores, you can use `Project` partition.

Figure 25: The cluster resource limit

3.1 Compile and Run

In fact, my execution strictly follows the instruction in the readme file of the given template.

3.1.1 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
```

- **bonus (command line application):**

```
mpic++ ./src/bonus.cpp -o bonus -fopenmp -std=c++11
```

- **bonus (GUI application):**

```
mpic++ ./src/bonus.cpp -o bonusg -fopenmp -I/usr/include -L/usr/local/lib
-L/usr/lib -lglut -lGLU -lGL -lm -O2 -DGUI -std=c++11
```

3.1.2 run

- **Sequential**

```
./seq $n_body $n_iterations
```

- **MPI**

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

- **Pthread**

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

- **CUDA**

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

- **OpenMP**

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

- **bonus**

```
mpirun -np $n_processes ./bonus $n_body $n_iterations $n_omp_threads
```

3.2 makefile

Makefile helps you simplify compilation command.

```
    make $command
```

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

When you need to recompile, please first run ‘make clean‘!

3.3 Sbatch script

MPI

For MPI program, you can use

```
#!/bin/bash
#SBATCH --job-name=your_job_name # Job name
#SBATCH --nodes=1                # Run all processes on a single node
#SBATCH --ntasks=20              # number of processes = 20
#SBATCH --cpus-per-task=1 # Number of CPU cores allocated to each process
#SBATCH --partition=Project      # Partition name: Project or Debug

cd /nfsmnt/119010355/CSC4005_2022Fall_Demo/project3_template/
mpirun -np 4 ./mpi 1000 100
mpirun -np 20 ./mpi 1000 100
mpirun -np 40 ./mpi 1000 100
```

Pthread

For pthread program, you can use

```
#!/bin/bash
#SBATCH --job-name=your_job_name # Job name
#SBATCH --nodes=1                # Run all processes on a single node
#SBATCH --ntasks=1               # number of processes = 1
#SBATCH --cpus-per-task=20 # Number of CPU cores allocated to each process
#SBATCH --partition=Project      # Partition name: Project or Debug

cd /nfsmnt/119010355/CSC4005_2022Fall_Demo/project3_template/
./pthread 1000 100 4
./pthread 1000 100 20
./pthread 1000 100 40
./pthread 1000 100 80
./pthread 1000 100 120
./pthread 1000 100 200
...
```

here you can create as many threads as you want while the number of cpu cores are fixed.

CUDA

For CUDA program, you can use

```
#!/bin/bash

#SBATCH --job-name CSC3150CUDADemo ## Job name
#SBATCH --gres=gpu:1             ## Number of GPUs required for job execution.
#SBATCH --output result.out      ## filename of the output
#SBATCH --partition=Project      ## the partitions to run in (Debug or Project)
#SBATCH --ntasks=1               ## number of tasks (analyses) to run
#SBATCH --gpus-per-task=1        ## number of gpus per task
#SBATCH --time=0-00:02:00        ## time for analysis (day-hour:min:sec)

## Run the script
srun ./cuda 10000 100
```

openmp

For openmp program, you can use

```
#!/bin/bash

#SBATCH --job-name job_name ## Job name
#SBATCH --output result.out      ## filename of the output
```

```
#SBATCH --partition=Project        ## the partitions to run in (Debug or Project)
#SBATCH --ntasks=1                 ## number of tasks (analyses) to run
#SBATCH --gpus-per-task=1          ## number of gpus per task
#SBATCH --time=0-00:02:00          ## time for analysis (day-hour:min:sec)

## Compile the cuda script using the nvcc compiler
## You can compile your codes out of the script and simply srun the executable file .
## Run the script
./openmp 10000 100 20
```

To submit your job, use

```
sbatch xxx.sh
```

4 Result & Analysis

First, I show the GUI output (take sequential case as an example, I run 100 iterations here).

Beginning:

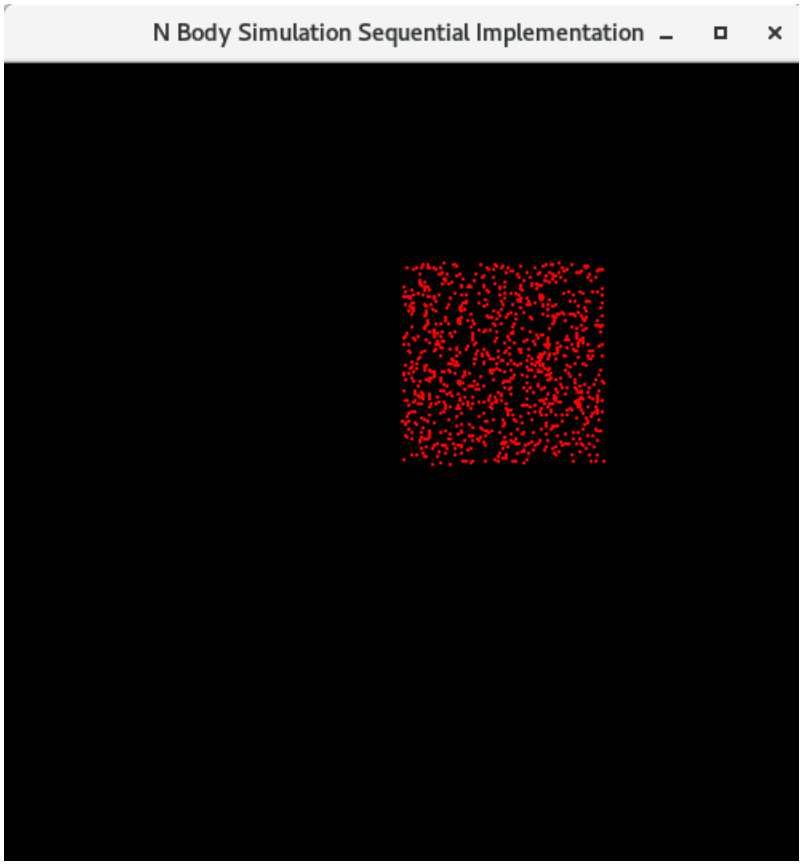


Figure 26: The beginning GUI output of sequential version

Running:

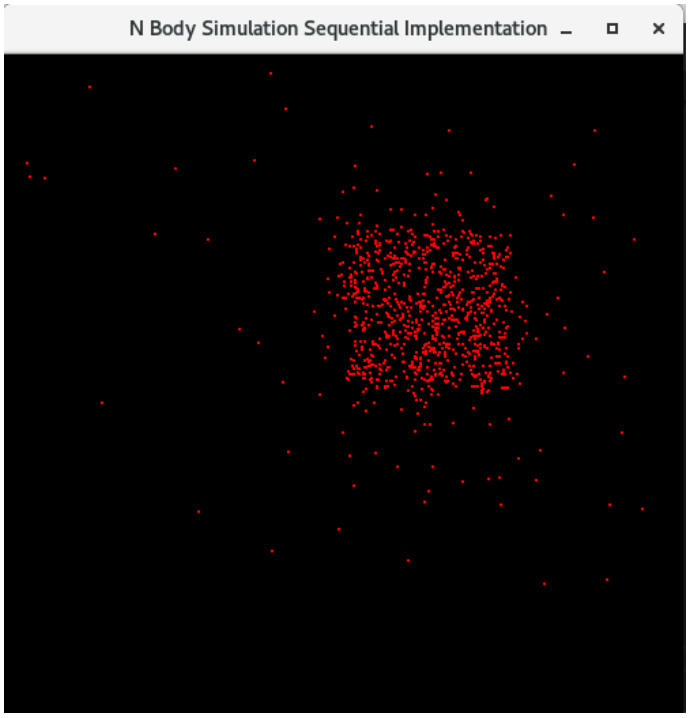


Figure 27: The running GUI output of sequential version

After 100 iterations:

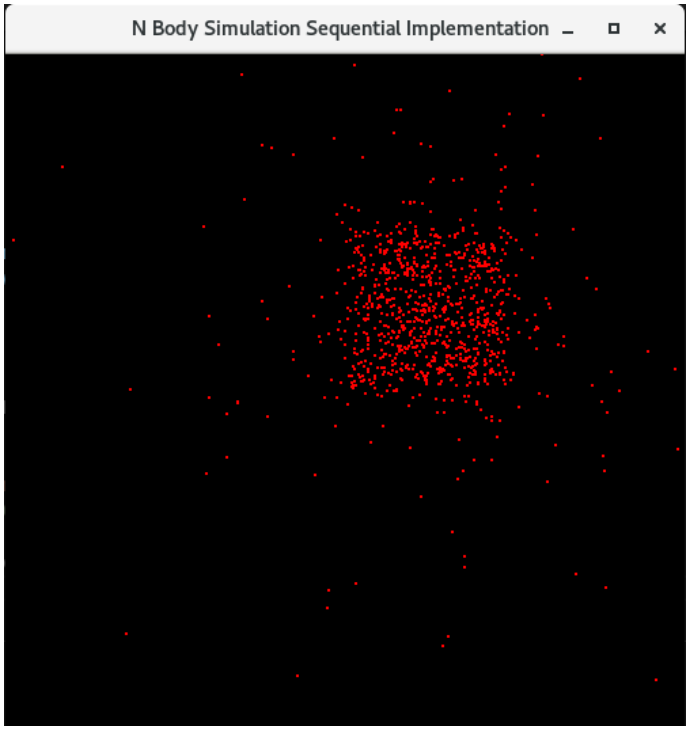
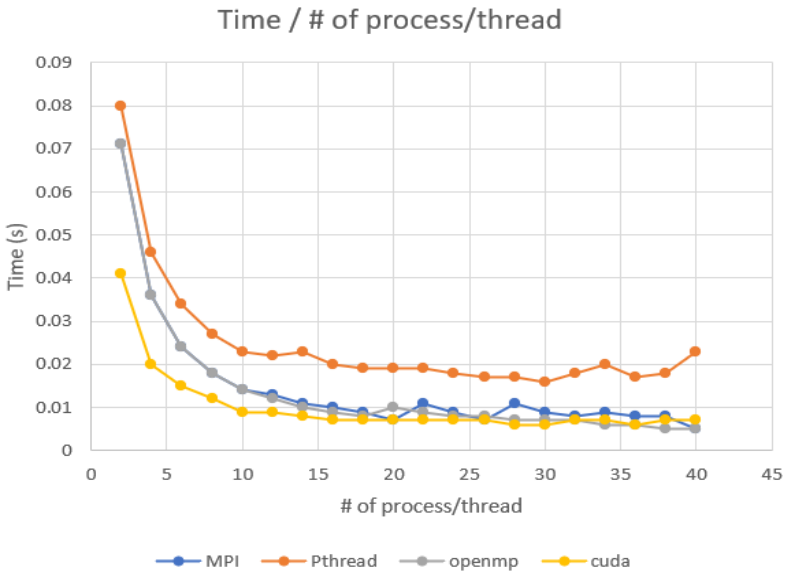


Figure 28: The GUI output of sequential version after 100 iterations

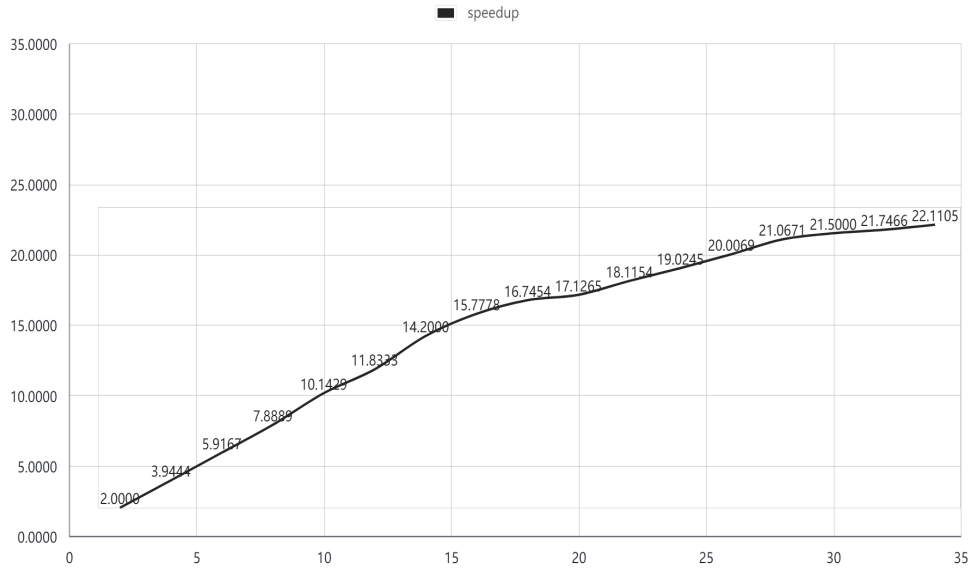
4.1 Comparison among different process/thread numbers

In this section we consider all five versions with different number of processes/threads. Here we fix the number of bodies to 1000, and analysize the image where y-axis represents the running time of one iteration and x-axis represents the number of processes/threads. As we can observe from the figure, as the number of processes/threads increases, we can significantly reduce the time cost of the same program task. This is because that regarding the same amount of data, we let multiple cores to process the data simultaneously. The larger the problem scale is, the more amount of time we can reduce by using multiple cores.



Then we seek to analysize the speedup of program w.r.t different number of processes/threads. Here we still fix the body numebr to 1000 and consider the OpenMP version, by the definition of the speedup, we can compute the speed up of the program in terms of different number of processes/threads.

$$S(n) = \frac{T_{sequential}}{T_{parallel}} \qquad (speedup \quad definition)$$



As we can see in the figure, the speedup does not conform a linear relationship with the process/thread number. This phenomenon can be explained by the Amdahl's Law.

$$S(n) = \frac{W_s + W_p}{W_s + W_p/n} = \frac{f + (1-f)}{f + \frac{1-f}{n}} = \frac{n}{1 + f(n-1)} \qquad (Amdahl's\ \ Law)$$

n: amount of cores
Ws: sequential workload
Wp: parallel workload
f: Ws/W (the proportion of sequential workload)

According to the Amdahl's Law, As n (which represents the process/thread number) approaches infinity, the value of speedup will become 1/f, where f is the proportion of sequential workload. In other words, the speedup will not increases linearly along with the core amount. Instead, the asymptotic line of the speedup will be $S_n = \frac{1}{f}$

$$S(n) = \frac{W_s + W_p}{W_s + W_p/n} = \frac{f + (1-f)}{f + \frac{1-f}{n}} = \frac{n}{1 + f(n-1)} \Longrightarrow \frac{1}{f} \quad if \quad n \to \infty \quad (Amdahl's\ Law)$$

From the above figure, we can observe that the asymptotic line is $S(n) = 23$, thus we can have a guess that the sequential work proportion in 1000 bodies is $\frac{1}{23}$.

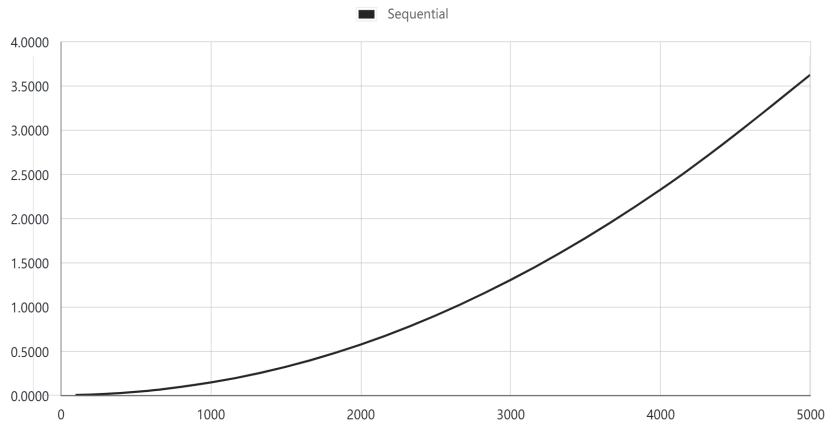
4.2 Comparison among different body numbers

In this section, we simply consider the sequential version of N-body simulation. Note that, For images in this subsection, all y-axis represents the running time of one iteration (here we choose the mode of the running time in each iteration as the time to prevent the influence of abnormal data) while x-axis represents the body numbers.

Assume the average computation time (computation time for velocity plus computation time

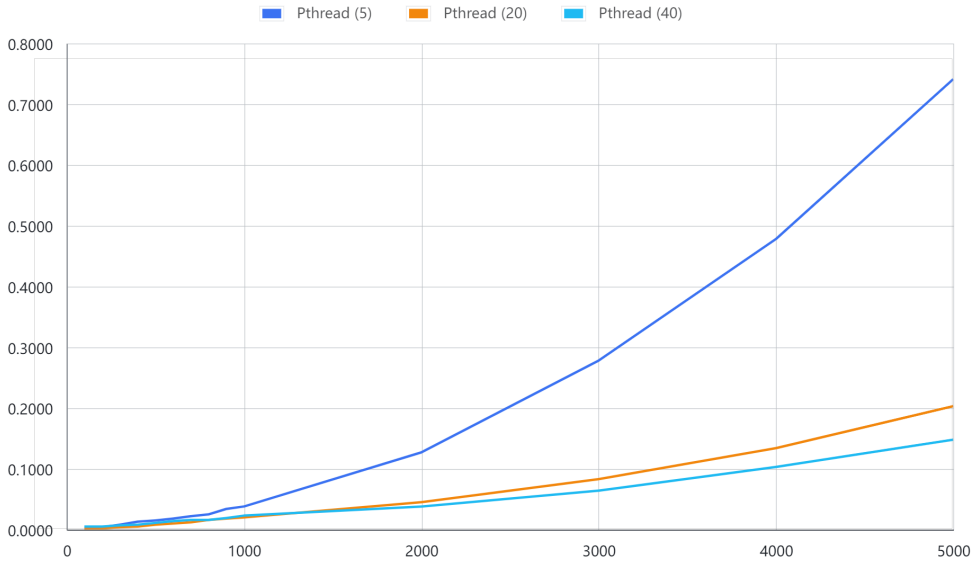
for position) for each body is identical, if we double the body number, the running time increases to four times. In this way, we can estimate the time complexity of N-body simulation in one iteration.

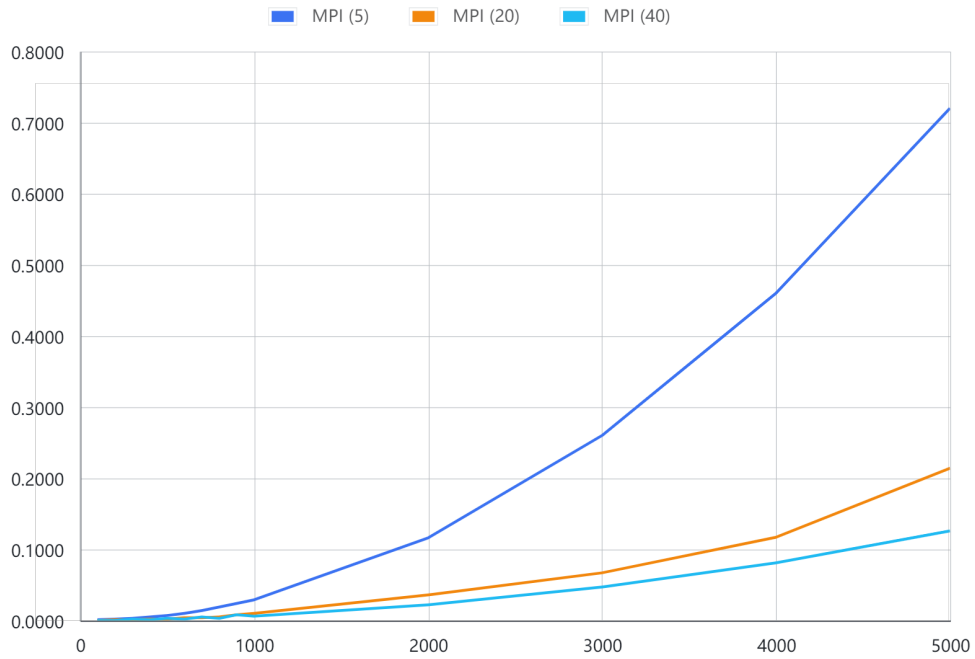
$$complexity = O(n^2) \qquad (time\ complexity)$$

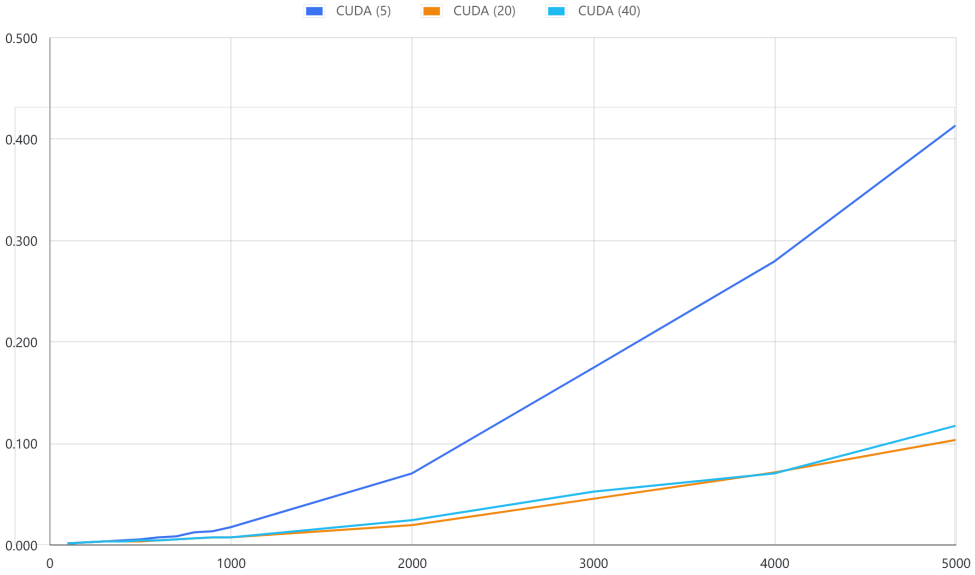


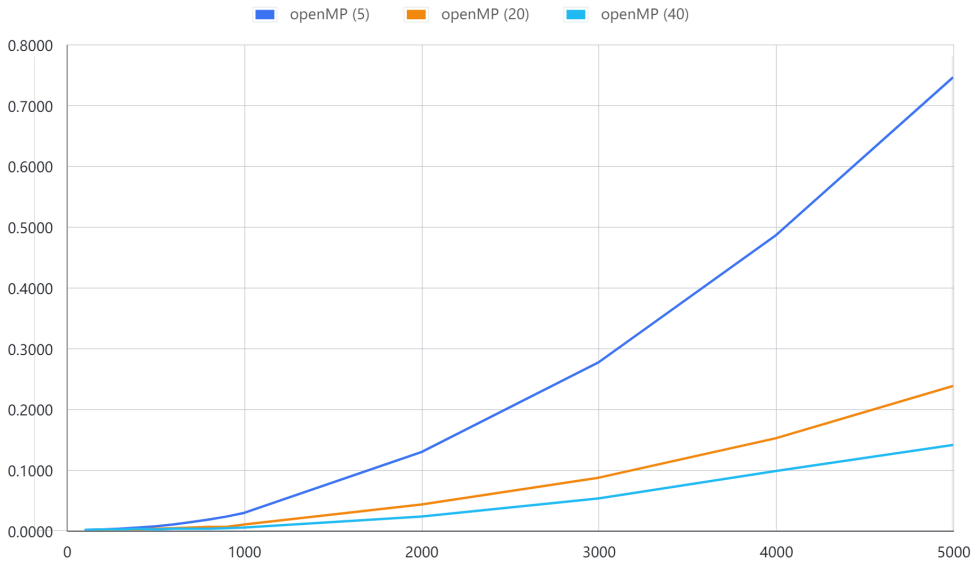
As we can see from the result, the sequential version of N-body simulation satisfies the n-square time complexity.

4.3 Comparison among the influence of multithreads and multiporcesses




MPI (5)
MPI (20)
MPI (40)
0.8000
0.7000
0.6000
0.5000
0.4000
0.3000
0.2000
0.1000
0.0000
0
1000
2000
3000
4000
5000


CUDA (5)
CUDA (20)
CUDA (40)
0.500
0.400
0.300
0.200
0.100
0.000
0
1000
2000
3000
4000
5000



In this section we compare the influence of multiporcesses and multithreads to the program. Here we draw the image for the MPI, pthread, openMP, and CUDA version of N-body simulation. In our images, the y-axis denotes the running time of one iteration and x-axis denotes the number of bodies. And we also plot 3 cases where the threads/processes number is set to be 5, 20, and 40, respectively. As we can observe from the figures above, as number of bodies increases, the time consumptions of the program of all implementations increase. However, the increasing slope is the minimal for CUDA version, and the maximal for openMP version. The different increase slpoe may comes from different implementation of multiporcesses and multithreads, which is not the main focus of our discussion.

We mainly focus on the increase of running time by multiporcesses and multithreads. Note that, the MPI version uses multiporcesses and the other three version uses multithreads. We can observe from the image that, when the process/thread number increases from 5 to 20, the running time increase very rapidly. However, when the process/thread number increases from 20 to 40, the increase of running time is not much significant. For multiporcesses, the reason is already explained by the Amdahl's law as mentioned above. But for multithreads, why the decrease of running time is much slower if threads number is relatively large? Here we consider a simple case, if we run a program on a personal comupter, can we decreases the running time to zero when we increases the thread number to a very large value? Obviously no. Each computer has a CPU containing many cores, each core has a maximum number of threads running on it which is determined by the hardware. If the thread number is larger than the maximum number, threads will race to get the position on CPU cores. In this case, the running time is not improved as the thread number increases. What's worse, because many threads use the resources of CPU alternatively, the running time for each thread may become lower(not necessary, depends on specific situations). Then we consider a program running on the HPC, compared to a personal computer, HPC cluster has many CPUs, thus more threads can run simultaneously on the cluster. However, when the thread number become

hundreds, the case in which threads compete for CPU resources can still happen. This may be the reason why the decrease of running time is much slower if thread number is relatively large.

Another interesting observation is that OpenMP, pthread, and CUDA all use multithreads, but the time decrease with respect to the three versions is not identical. This may be resulted from many aspects. One aspect that worths mentioned may be the efficiency of threads. In the CUDA implementation, each thread's workload is predefined (i.e each thread computes a defined number of bodies). In this case, the efficiency of threads may lose and time may be wasted because the computation time for each body may not be identical (although we assume it is identical in 3.1 for convernience), the stragglers will drag the running time. The time that each thread finishes their job is asynchronous and the final running time is determined by the thread which uses most time. In our pthread implementation, we adopt dynamic allocation, which means all threads keep working until all jobs are done. Therefore, a little increase in thread numbers may contribute to large running time improvements. This phenomenon can also be obverved from the image. Consider the number of bodies equal to 5000, in the pthread version, when the thread number increases from 5 to 20, the running time in one iteration decreases from 0.75 to 0.2; in the CUDA version, when the thread number increases from 5 to 20, the running time in one iteration decreases from 0.4 to 0.1, the decrease magnitude is much smaller.

5 Conclusion

Here I will summarize what I have learned when writing assignment 3.

- Learn about parallel computing. For example, different processes share different memory, and different threads can share the same global variables. The same pointer cannot be valid all the time. Both CPU and GPU in CUDA have visible regions.
- The problems to be noticed when writing parallel programs (process synchronization, process communication, thread creation and join, cost, data race)
- MPI & Pthread & openmp & CUDA API usage.
- I learned how to design experiments to compare different implementation in several dimensions.
- I can manipulate processes and threads more flexibly. For example, using threads to implement dynamic scheduling, and deal with the remainder case for multiple processes.

That's all!!!

Bibliography

Wikipedia (2022). *N-body simulation*. URL: https://en.wikipedia.org/wiki/N-body_simulation (visited on 14th Oct. 2022).