CSC4005HW3: N-body Simulastion & Performance Analysis

118010202 / LIU Zhixuan

November 17, 2021

1 Abstract

In physics and astronomy, an N-body simulation is a simulation of a dynamical system of particles, usually under the influence of physical forces, such as gravity (see n-body problem). N-body simulations are widely used tools in astrophysics, from investigating the dynamics of few-body systems like the Earth-Moon-Sun system to understanding the evolution of the large-scale structure of the universe.[1] In physical cosmology, N-body simulations are used to study processes of non-linear structure formation such as galaxy filaments and galaxy halos from the influence of dark matter. Direct N-body simulations are used to study the dynamical evolution of star clusters. And the interaction among each bodies is $F = \frac{GM_1M_2}{r^2}$.

In this assignment, I performed both parallel version and sequential version to draw the Mandelbrot Set using ImGui; moreover, for the parallel implementation, I use both MPI method and pthread method, OpenMp method, MPI&OpenMP method, and cuda method to realize this simulation. At last, the corresponding experiments and some efficient analysis is performed on all the algorithms.

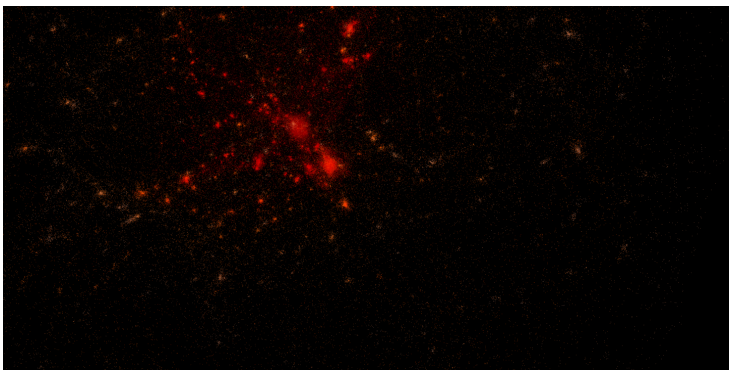

Figure 1: Illustration of N-body Simulation

2 Introduction to N-body Simulation and Sequential Realization

In this section, we briefly introduce N-body Simulation and how to implement the sequential version of it using C++ and some complexity analysis of it.

Our goal is to initialize N bodies first, and then simulate the movement of these N bodies according to the basic principles of physics. It mainly follows these two formulas: $F = \frac{GM_1M_2}{r^2}$ and $F = ma$. We need to traverse all the bodies, and find the gravitational force between the two, and calculate the acceleration of each one, and depict the movement of the bodies, and finally draw it on the GUI.

Now, let's talk about the code implementation of this sequential algorithm using C++. In the following codes, i use ImGui to implement the drawing process.

Before the simulation, I receive some basic information to initialize the canvas. These are the default initialization of the canvas, which represent how many bodies we have, the size of the canvas, and some useful parameter.

```
static float gravity = 100;
static float space = 800;
static float radius = 5;
static int bodies = 200;
static float elapse = 0.001;
static ImVec4 color = ImVec4(1.0f, 1.0f, 0.4f, 1.0f);
static float max_mass = 50;
```

Firstly, I define a buffer, which is called BodyPool::pool, to store the information of the N-body. All its attributes are all vectors, because they store the information of all bodies, and the information of each body is determined by the index. Where x,y represent all the coordinates of bodies in a rectangular coordinate system. vx and vy are the speed, ax and ay are the accelerate. All the bodies information are set randomly and the bodies are set to at rest at the beginning.

```
std::vector<double> x;
std::vector<double> y;
std::vector<double> vx;
std::vector<double> vy;
std::vector<double> ax;
std::vector<double> ay;
std::vector<double> m;
```

Now let's move into the main loop. The first thing we do is to check whether the basic parameters have been changed or not in this iteration.

```
if (current_space != space || current_bodies != bodies || current_max_mass !=
    max_mass) {
    space = current_space;
    bodies = current_bodies;
    max_mass = current_max_mass;
    pool = BodyPool{static_cast<size_t>(bodies), space, max_mass};
}
```

We now go into the main function:

```
pool.update_for_tick(elapse, gravity, space, radius);
void update_for_tick(double elapse,
                     double gravity,
                     double position_range,
                     double radius) {
    ax.assign(size(), 0);
    ay.assign(size(), 0);
    for (size_t i = 0; i < size(); ++i) {
        for (size_t j = i + 1; j < size(); ++j) {
            check_and_update(get_body(i), get_body(j), radius, gravity);
```

```
        }
    }
    for (size_t i = 0; i < size(); ++i) {
        get_body(i).update_for_tick(elapse, position_range, radius);
    }
}
```

In order to compare between each of two bodies, first, it go through each bodies i, and in each iteration of i, it go through each element j which is bigger than i. Then it calculates the distance, and the attraction between the target i and all the j, and also calculate the according acceleration and store the acceleration into the information buffer ax, ay. And also according to the time delta, the velocity will also be changed.

And also we will consider the situation when two bodies collide with each other.

```
static void check_and_update(Body i, Body j, double radius, double gravity) {
    auto delta_x = i.delta_x(j);
    auto delta_y = i.delta_y(j);
    auto distance_square = i.distance_square(j);
    auto ratio = 1 + COLLISION_RATIO;
    if (distance_square < radius * radius) {
        distance_square = radius * radius;
    }
    auto distance = i.distance(j);
    if (distance < radius) {
        distance = radius;
    }
    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_vx() -= scalar * delta_x * j.get_m();
        i.get_vy() -= scalar * delta_y * j.get_m();
        j.get_vx() += scalar * delta_x * i.get_m();
        j.get_vy() += scalar * delta_y * i.get_m();
        // NOW RELAX THE DISTANCE A BIT: AFTER THE COLLISION, THERE MUST BE
        // AT LEAST (RATIO * RADIUS) BETWEEN THEM
        i.get_x() += delta_x / distance * ratio * radius / 2.0;
        i.get_y() += delta_y / distance * ratio * radius / 2.0;
        j.get_x() -= delta_x / distance * ratio * radius / 2.0;
        j.get_y() -= delta_y / distance * ratio * radius / 2.0;
    } else {
        // UPDATE ACCELERATION ONLY WHEN NO COLLISION
        auto scalar = gravity / distance_square / distance;
        i.get_ax() -= scalar * delta_x * j.get_m();
        i.get_ay() -= scalar * delta_y * j.get_m();
        j.get_ax() += scalar * delta_x * i.get_m();
        j.get_ay() += scalar * delta_y * i.get_m();
    }
}
```

After updating the information and check whether there will be collision happen, next we will completely update the information of the target i one by one for this iteration using the information we calculated above using the function above.

```
void update_for_tick(
        double elapse,
        double position_range,
        double radius) {
```

```
    get_vx() += get_ax() * elapse;
    get_vy() += get_ay() * elapse;
    handle_wall_collision(position_range, radius);
    get_x() += get_vx() * elapse;
    get_y() += get_vy() * elapse;
    handle_wall_collision(position_range, radius);
}
```

After update all the information in this iteration, we now comes to the final part, which is drawing all the new updates on the ImGui.

Basically, the main idea to draw the things is to go through each body, which is represented by i in each iteration, and then draw it on the ImGui.

```
for (size_t i = 0; i < pool.size(); ++i) {
    auto body = pool.get_body(i);
    auto x = p.x + static_cast<float>(body.get_x());
    auto y = p.y + static_cast<float>(body.get_y());
    draw_list->AddCircleFilled(ImVec2(x, y), radius, ImColor{color});
}
}
```

Now, let's evaluate the complexity of sequential N-body simulation computation algorithm. The computational complexity : $O(N^2)$ where, N the number of the bodies. This is because every time we have to compare the relationship between the two bodies, while traversing the bodies, we must traverse and calculate the relationship with other bodies.

3 Parallel N-body Simulation Computation using MPI

It is obvious that sequential N-body simulation computation does not have a very good performance in time complexity. However, this algorithm can be much more efficient if it is distributed computed, therefore, the computational time will be saved. The first idea of doing this is to use MPI method to spead up the process, and the main idea is as follows:

In the MPI algorithm, firstly, the assignment will be given and initialized in the master process. Then the master process will distribute the information in the slave processes, each slave process is equally assigned to calculate some part of the n-bodies, and only up date its information. After calculate and update their own parts, they will sent the result to the master process. And the master process will write the information in the buffer "pool", and then using ImGui to visualize the body moving.

The process for this distributed computing is shown in the figure 2 below:

Now let's look at how to implement this process using C++.

Similar to the sequential process, we first initiate a structure called pool, which stored the information of all the bodies in the master process, and initialize some necessary parameters.

Next, in the MASTER process, the MASTER will send the parameter information to all the slave process by using iteration:

```
for(int i=0; i<slave_size; i++){
   MPI_Send(&(para[0]), 4, MPI_INT, (i+1), (i+1)*10+8, MPI_COMM_WORLD);
   MPI_Send(&(paraf[0]), 3, MPI_INT, (i+1), (i+1)*10+9, MPI_COMM_WORLD);
   MPI_Send(&(c_x[0]), 1, MPIVector, (i+1), (i+1)*10+1, MPI_COMM_WORLD);
   MPI_Send(&(c_y[0]), 1, MPIVector, (i+1), (i+1)*10+2, MPI_COMM_WORLD);
   MPI_Send(&(c_vx[0]), 1, MPIVector, (i+1), (i+1)*10+3, MPI_COMM_WORLD);
   MPI_Send(&(c_vy[0]), 1, MPIVector, (i+1), (i+1)*10+4, MPI_COMM_WORLD);
```

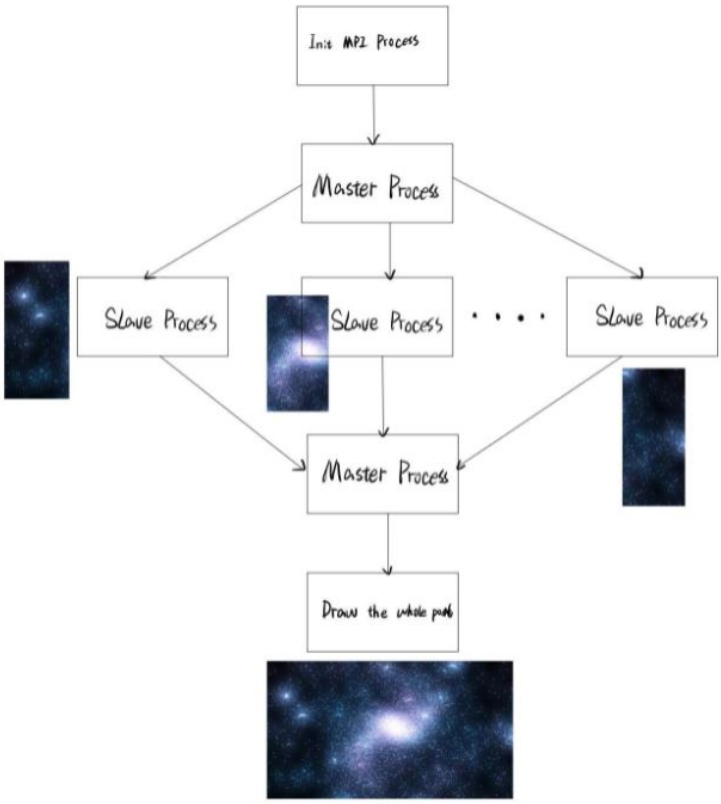

Figure 2: Process of Parallel computation using MPI

```
8     MPI_Send(&(c_ax[0]), 1, MPIVector, (i+1), (i+1)*10+5, MPI_COMM_WORLD);
9     MPI_Send(&(c_ay[0]), 1, MPIVector, (i+1), (i+1)*10+6, MPI_COMM_WORLD);
10    MPI_Send(&(c_m[0]), 1, MPIVector, (i+1), (i+1)*10+7, MPI_COMM_WORLD);
11 }
```

Then, in the slave process, where all the computations are distributed, slave first receieve all the information sent by MASTER process:

```
1 MPI_Recv(&(s_pool.x[0]), 1, MPILocalVector,0,rank*10+1,MPI_COMM_WORLD, &status);
2 MPI_Recv(&(s_pool.y[0]), 1, MPILocalVector,0,rank*10+2,MPI_COMM_WORLD, &status);
3 MPI_Recv(&(s_pool.vx[0]), 1, MPILocalVector,0,rank*10+3,MPI_COMM_WORLD, &status);
4 MPI_Recv(&(s_pool.vy[0]), 1, MPILocalVector,0,rank*10+4,MPI_COMM_WORLD, &status);
5 MPI_Recv(&(s_pool.ax[0]), 1, MPILocalVector,0,rank*10+5,MPI_COMM_WORLD, &status);
6 MPI_Recv(&(s_pool.ay[0]), 1, MPILocalVector,0,rank*10+6,MPI_COMM_WORLD, &status);
7 MPI_Recv(&(s_pool.m[0]), 1, MPILocalVector,0,rank*10+7,MPI_COMM_WORLD, &status);
```

Then, only calculate part of the bodies and store the body information in the local buffer, the culculation process is the same as the sequential process:

```
1 s_pool.update_for_tick(s_paraf[0], s_paraf[1], s_para[0], s_paraf[2], rank,
      local_bodies, s_bodies);
```

After calculate their own part, the slave will send the buffer to the MASTER process, and the slave process finishes its job.

```
1 MPI_Isend(&(s_pool.x[0]), 1, MPILocalVector, 0, 1, MPI_COMM_WORLD, &request);
2 MPI_Isend(&(s_pool.y[0]), 1, MPILocalVector, 0, 2, MPI_COMM_WORLD, &request);
3 MPI_Isend(&(s_pool.vx[0]), 1, MPILocalVector, 0, 3, MPI_COMM_WORLD, &request);
4 MPI_Isend(&(s_pool.vy[0]), 1, MPILocalVector, 0, 4, MPI_COMM_WORLD, &request);
5 MPI_Isend(&(s_pool.ax[0]), 1, MPILocalVector, 0, 5, MPI_COMM_WORLD, &request);
```

```
MPI_Isend(&(s_pool.ay[0]), 1, MPILocalVector, 0, 6, MPI_COMM_WORLD, &request);
MPI_Isend(&(s_pool.m[0]), 1, MPILocalVector, 0, 7, MPI_COMM_WORLD, &request);
```

Then, the MASTER process receive the buffer from the slave processes, and write the information to the pool:

```
MPI_Recv(&(c_x[0]), 1, MPIVector,MPI_ANY_SOURCE, 1, MPI_COMM_WORLD,&status);
    slave_rank = status.MPI_SOURCE;
    for (int j=0; j<local_bodies; j++){
        if((slave_rank-1)*local_bodies + j < bodies){
            pool.x[(slave_rank-1)*local_bodies + j] = c_x[j];
        }
    }
```

The last part is for canvas to draw the information using ImGui:

```
for (size_t i = 0; i < pool.size(); ++i) {
    auto body = pool.get_body(i);
    auto x = p.x + static_cast<float>(body.get_x());
    auto y = p.y + static_cast<float>(body.get_y());
    draw_list->AddCircleFilled(ImVec2(x, y), radius, ImColor{color});
}
```

This is pretty much about the concise implementation of the parallel N-body simulation Computation using MPI.

As for the computational complexity, since the whole mathematical computational algorithm does not change, the only change is that the whole process is distributed to p slave processes to compute, therefore, the computational complexity can be written as in our cases are $O(\frac{N^2}{p})$.

4 Parallel N-body Simulation Computation using Pthread

It is obvious that sequential Mandelbrot Set computation does not have a very good performance in time complexity. However, this algorithm can be much more efficient if it is distributed computed, therefore, the computational time will be saved. The second idea of doing this is to use Pthread method to spead up the process, and the main idea is as follows:

In the Pthread algorithm, firstly, the assignment will be given and initialized in the main function. Then the main function will fork several child processes, and will distribute the information in the child processes, each child process is equally assigned to on part of the whole picture, and their aim is to draw their own part of the picture. Each child process will write the information in the buffer "pool". After writing the information, the each thread will join together, and then using ImGui to visualize the drawing.

The process for this distributed computing is shown in the figure 3 below:

Now let's look at how to implement this Pthread process using C++.

Similar to the sequential process, we first initiate a structure called pool, which stored the information of the drawing based on the pixel in the master process, and initialize some necessary parameters.

Next, in the main function process, we will create a space to store some threads, based on the given threads number:

```
pthread_t threads[THREADS_NUM];
```

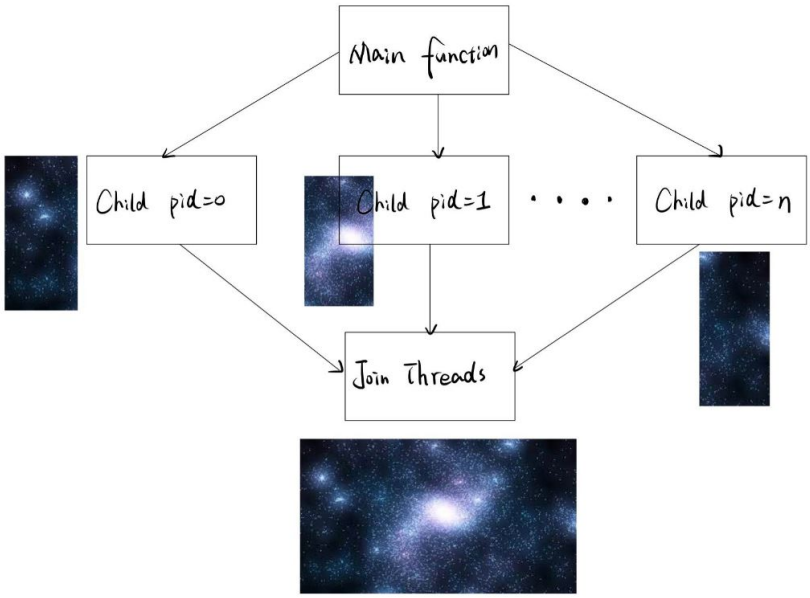

Figure 3: Process of Parallel computation using Pthread

Then, we also create a structure called argument, which will save all the information that each thread's function will receive, and create and send the information to thread:

```
struct Arguments{
    BodyPool *pool;
    double elapse;
    double gravity;
    double position_range;
    double radius;
    int pid;
    int local_bodies;
    int bodies;
};

for(int m=0; m<THREADS_NUM; m++){
    pthread_create(&threads[m], nullptr, local_process, new Arguments{
        .pool = &pool,
        .elapse = elapse,
        .gravity = gravity,
        .position_range = space,
        .radius = radius,
        .pid = m,
        .local_bodies = local_bodies,
        .bodies = bodies
    });
}
}
```

Then, int the child process, firstly, they receive the information, and then only calculate part of the image and store the image information in the pool. To let them all write to the pool correctly, I passed the pointer of the canvas so that each thread can find the address to write to:

```
void *local_process(void *arg_ptr){
    auto arguments = static_cast<Arguments *>(arg_ptr);
    // PTHREAD_MUTEX_LOCK(&MUTEX_P);
```

```
    (*arguments->pool).update_for_tick(arguments->elapse, arguments->gravity,
                       arguments->position_range, arguments->radius,
                       arguments->pid+1, arguments->local_bodies,
                       arguments-> bodies);
    // PTHREAD_MUTEX_UNLOCK(&MUTEX_P);
    delete arguments;
    return nullptr;
}
```

After calculate their own part, the all the threads will join together, and the whole process ends.

```
for(auto & n : threads){
    pthread_join(n, nullptr);
}
pthread_attr_destroy(&attr);
```

The last part is for pool to draw the information using ImGui:

```
for (size_t i = 0; i < pool.size(); ++i) {
    auto body = pool.get_body(i);
    auto x = p.x + static_cast<float>(body.get_x());
    auto y = p.y + static_cast<float>(body.get_y());
    draw_list->AddCircleFilled(ImVec2(x, y), radius, ImColor{color});
}
```

This is pretty much about the concise implementation of the parallel N-body Simulation Computation using Pthread.

As for the computational complexity, since the whole mathematical computational algorithm does not change, the only change is that the whole process is distributed to c child processes to compute, therefore, the computational complexity can be written as in our cases are $O(\frac{N^2}{c})$.

5 Parallel N-body Simulation Computation using OpenMP

It is obvious that sequential Mandelbrot Set computation does not have a very good performance in time complexity. However, this algorithm can be much more efficient if it is distributed computed, therefore, the computational time will be saved. The third idea of doing this is to use OpenMP method to spead up the process, and the main idea is as follows:

In the OpenMP algorithm, similar to the Pthread algorithm, the only difference is the way it implement. Instead of fork child process in using pthread, it is used in this way.

```
{
omp_set_num_threads(THREADS_NUM);
pool.update_for_tick(elapse,gravity,space,radius, bodies);
}

void update_for_tick(double elapse,
                     double gravity,
                     double position_range,
                     double radius,
                     int bodies) {
    int j;
    #pragma omp parallel private(j)
    {
```

```
    // INT THREAD_ID = OMP_GET_THREAD_NUM();
    ax.assign(size(), 0);
    ay.assign(size(), 0);
    #pragma omp for schedule(static)
    for (int i = 0; i < bodies; ++i) {
        if(i<bodies){
            for (int j = i + 1; j < bodies; ++j) {
                if(j<bodies){
                    check_and_update(get_body(i), get_body(j), radius, gravity);
                }
            }
        }
    }
    #pragma omp for schedule(static)
    for (int i=0; i < bodies; ++i) {
        if(i<bodies){
            get_body(i).update_for_tick(elapse, position_range, radius);
        }
    }
    }
}
```

The process for this distributed computing is shown in the figure 4 below:

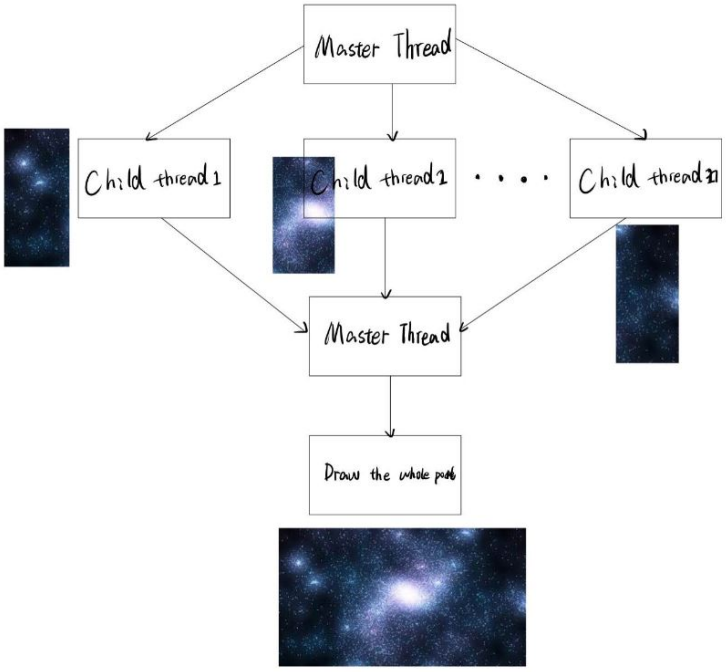

Figure 4: Process of Parallel computation using OpenMP

6 Bonus: Parallel N-body Simulation Computation using OpenMP and MPI

It is obvious that sequential Mandelbrot Set computation does not have a very good performance in time complexity. However, this algorithm can be much more efficient if it is distributed computed, therefore, the computational time will be saved. The third idea

of doing this is to use MPI and OpenMP method to spead up the process, and the main idea is as follows:

This algorithm is based on the MPI. After the MASTER process distribute jobs to the SLAVE process, in each SLAVE process, it can call the OpenMP to generate more threads in each SLAVE process. Therefore there will be more threads to implement the job, eache process get lesser jobs. The differences in codes are shown below, where the baseline is the MPI algorithm process.

```
{
omp_set_num_threads(THREADS_NUM);
pool.update_for_tick(elapse,gravity,space,radius, bodies);
}

void update_for_tick(double elapse,
                     double gravity,
                     double position_range,
                     double radius,
                     int bodies) {
    int j;
    #pragma omp parallel private(j)
    {
    // INT THREAD_ID = OMP_GET_THREAD_NUM();
    ax.assign(size(), 0);
    ay.assign(size(), 0);
    #pragma omp for schedule(static)
    for (int i = 0; i < bodies; ++i) {
        if(i<bodies){
            for (int j = i + 1; j < bodies; ++j) {
                if(j<bodies){
                    check_and_update(get_body(i), get_body(j), radius, gravity);
                }
            }
        }
    }
    #pragma omp for schedule(static)
    for (int i=0; i < bodies; ++i) {
        if(i<bodies){
            get_body(i).update_for_tick(elapse, position_range, radius);
        }
    }
    }
}
```

The process for this distributed computing is shown in the figure 5 below:

7 Parallel N-body Simulation Computation using Cuda

It is obvious that sequential Mandelbrot Set computation does not have a very good performance in time complexity. However, this algorithm can be much more efficient if it is distributed computed, therefore, the computational time will be saved. The second idea of doing this is to use Cuda method to spead up the process, and the main idea is as follows:

In the Cuda algorithm, firstly, the assignment will be given and initialized in the main function. Then the main function will fork several thread processes, and will distribute the information in the thread processes, each thread process is equally assigned to on

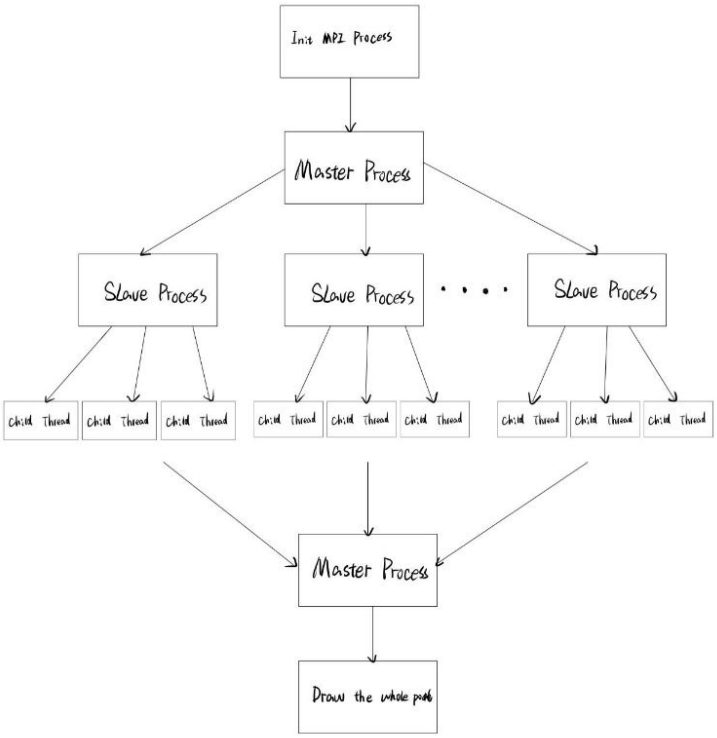

Figure 5: Process of Parallel computation using MPI and OpenMP

part of the n bodies, and their aim is to draw their own part of the bodies. Each thread process will write the information in the buffer "pool". After writing the information, the each thread will join together, and then using ImGui to visualize the drawing.

The process for this distributed computing is shown in the figure 6 below:

The major differences are std::vector cannot be used in the cuda process, so we mainly swift to array base implementation. The second thing is that the array initialization is in the main function, but cuda devices need to have access to it. Therefore, we use Cudamemcpy to let cuda devices to get access to the array and change on it.

```
1  double *hostX = new double[bodies];
2  double *hostY = new double[bodies];
3  double *hostVX = new double[bodies];
4  double *hostVY = new double[bodies];
5  double *hostAX = new double[bodies];
6  double *hostAY = new double[bodies];
7  double *hostM = new double[bodies];
8  double *x;
9  cudaMalloc(&x, sizeof(double) * bodies);
10 double *y;
11 cudaMalloc(&y, sizeof(double) * bodies);
12 double *vx;
13 cudaMalloc(&vx, sizeof(double) * bodies);
14 double *vy;
15 cudaMalloc(&vy, sizeof(double) * bodies);
16 double *ax;
17 cudaMalloc(&ax, sizeof(double) * bodies);
18 double *ay;
19 cudaMalloc(&ay, sizeof(double) * bodies);
20 double *m;
21 cudaMalloc(&m, sizeof(double) * bodies);
```

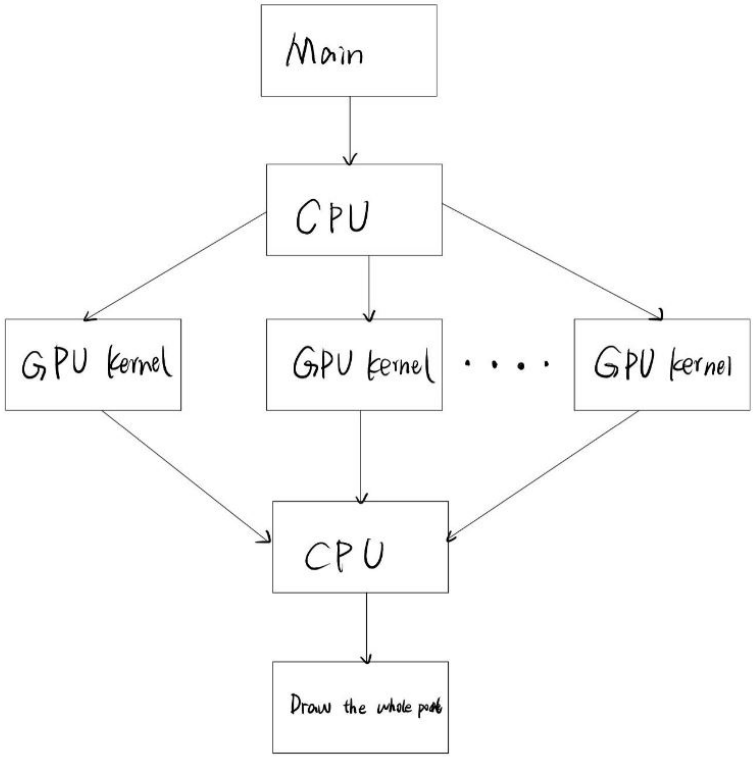

Figure 6: Process of Parallel computation using Cuda

```
22 initArray(hostX, hostY, hostM, bodies ,space, max_mass);
23 cudaMemcpy(x, hostX, sizeof(double)*bodies, cudaMemcpyHostToDevice);
24 cudaMemcpy(y, hostY, sizeof(double)*bodies, cudaMemcpyHostToDevice);
25 cudaMemcpy(vx, hostVX, sizeof(double)*bodies, cudaMemcpyHostToDevice);
26 cudaMemcpy(vy, hostVY, sizeof(double)*bodies, cudaMemcpyHostToDevice);
27 cudaMemcpy(ax, hostAX, sizeof(double)*bodies, cudaMemcpyHostToDevice);
28 cudaMemcpy(ay, hostAY, sizeof(double)*bodies, cudaMemcpyHostToDevice);
29 cudaMemcpy(m, hostM, sizeof(double)*bodies, cudaMemcpyHostToDevice);
```

This is pretty much about the concise implementation of the parallel N-body Simulation Set Computation using Cuda.

As for the computational complexity, since the whole mathematical computational algorithm does not change, the only change is that the whole process is distributed to t thread processes to compute, therefore, the computational complexity can be written as in our cases are $O(\frac{N^2}{t})$.

8 Compile and Run

I use the template on the blackboard to write the program, which use ImGui to display the image. So here are some command to run my code.

IMPORTANT: Please put my codes under file src and use the file name: "main.cpp". And put the "body.hpp" file to the under the /include/body folder, and also change the CMakeList.txt in each version of algorithm.

8.1 Build

For sequential, MPI, pthread, OpenMp, MPI + OpenMP, please do the following to build the file:

```
$ rm -rf build
$ mkdir build
$ cd build
$ cmake ..
$ make
```

as for the cuda version:

```
$ cd csc4005-imgui
$ mkdir build && cd build
$ source scl_source enable devtoolset-10
$ CC=gcc CXX=g++ cmake ..
$ make -j12
```

8.2 Run Sequential Program

To run my sequential codes, the sbatch shell codes are as follows:

```
#!/bin/bash
#SBATCH -J Zhixuan
#SBATCH -o sequential.out
#SBATCH -N 1
#SBATCH -n 1
#SBATCH -t 1

xvfb-run -a mpirun -np 1 csc4005_imgui
```

8.3 Run MPI Program

To run my MPI codes, the sbatch shell codes are as follows:

```
#!/bin/bash
#SBATCH -J Zhixuan
#SBATCH -o MPI_N1n4.out
#SBATCH -N 1
#SBATCH -n 4
#SBATCH -t 1

xvfb-run -a mpirun -np 4 csc4005_imgui
```

8.4 Run Pthread Program

To run my Pthread codes, the sbatch shell codes are as follows:

```
#!/bin/bash
#SBATCH -J Zhixuan
#SBATCH -o Pthread.out
#SBATCH -N 1
#SBATCH -n 1
#SBATCH -t 1

xvfb-run -a mpirun -np 1 csc4005_imgui
```

To test different thread, please change the thread number in the first line of the code in main.cpp, where you can modify the predefined THREADS_NUM.

8.5 Run OpenMP Program

To run my OpenMP codes, the sbatch shell codes are as follows:

```
#!/bin/bash
#SBATCH -J Zhixuan
#SBATCH -o OpenMP.out
#SBATCH -N 1
#SBATCH -n 1
#SBATCH -t 1

xvfb-run -a mpirun -np 1 csc4005_imgui 4
```

To test different thread, please change the thread number mentioned above, the default number will be 4.

8.6 Run MPI+OpenMP Program

To run my MPI+OpenMP codes, the sbatch shell codes are as follows:

```
#!/bin/bash
#SBATCH -J Zhixuan
#SBATCH -o MPI_N1n4.out
#SBATCH -N 1
#SBATCH -n 4
#SBATCH -t 1

xvfb-run -a mpirun -np 4 csc4005_imgui
```

To test different thread, please change the thread number in the first line of the code, where you can modify the predefined THREADS_NUM in the main.cpp.

8.7 Run Cuda Program

To run my OpenMP codes, the sbatch shell codes are as follows:

```
#!/bin/bash
#SBATCH -J Zhixuan
#SBATCH -o Cuda.out
#SBATCH -N 1
#SBATCH -n 1
#SBATCH -t 10

xvfb-run -a mpirun -np 1 csc4005_imgui 4
```

To test different cuda thread, please change the thread number mentioned above, the default number will be 4.

8.8 Sample Outcomes

Here are some sample outcomes for both sequential, mpi, and pthread, openmp, cuda algorithms.

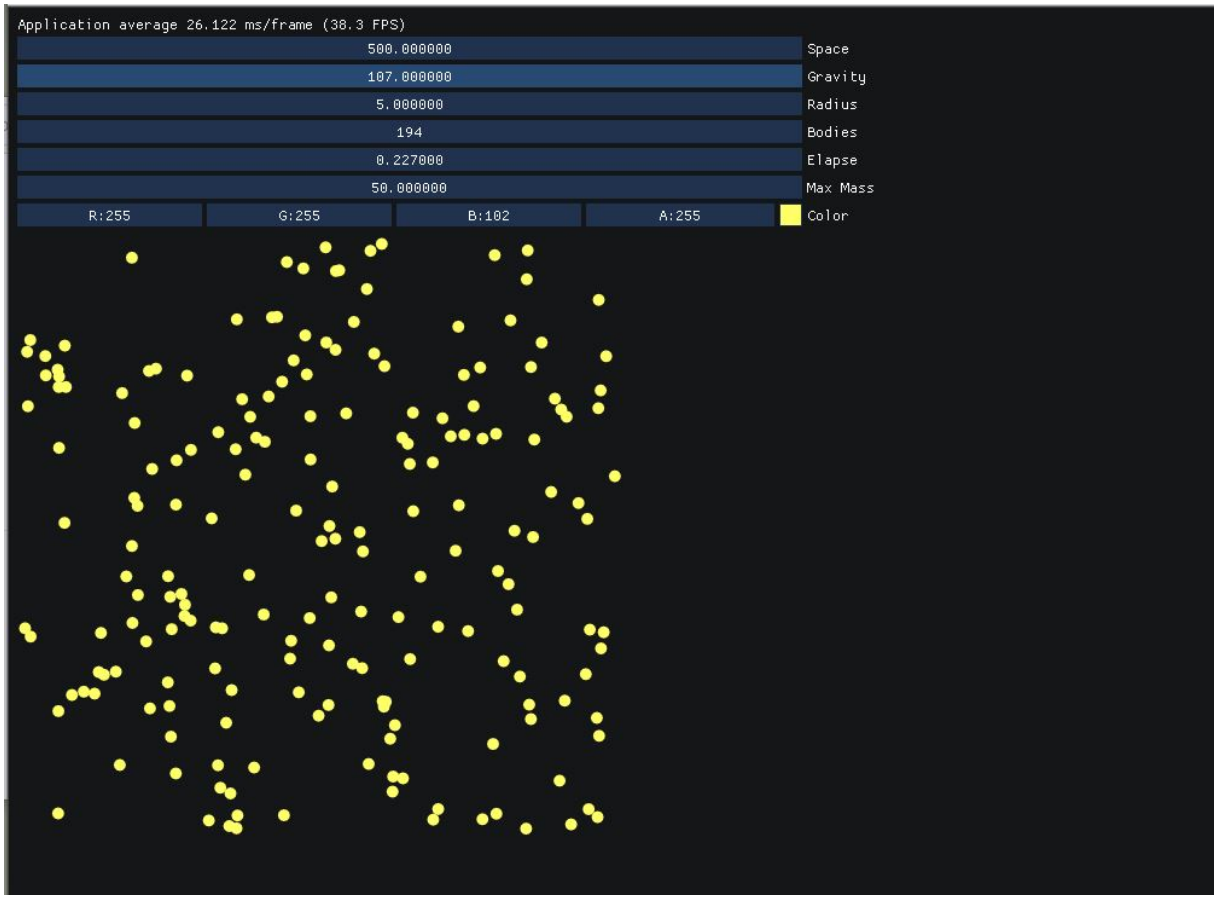

Figure 7: Sample GUI outcomes

9 Analysis

9.1 Comparison between Sequential, MPI, Pthread, MPI_Pthread and Cuda

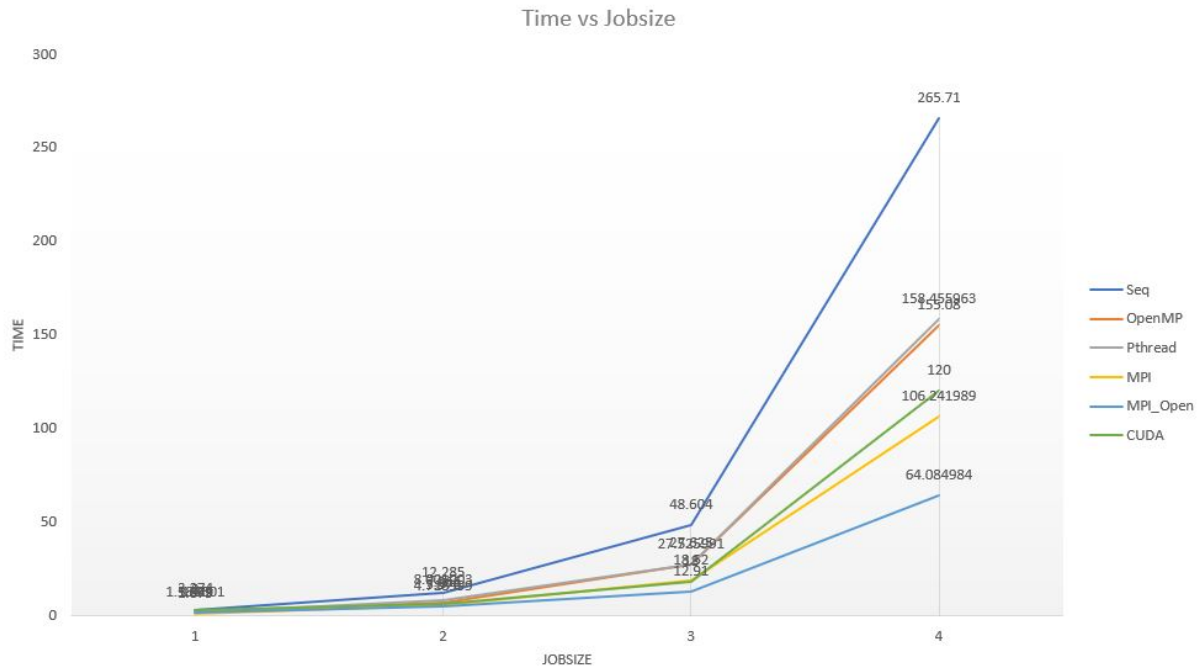

Figure 8: Comparison between all the algorithms

As the figure 8 shown above, when we fix the number of processes used by the program for MPI and Pthread (here we are talking about the fixed number, regardless of the number of masters in the mpi program, because when the MPI is executing the program, the MASTER program actually does not perform calculations, and It is to allocate tasks. In order to make the comparison more convincing, the number of processes mentioned here is slave process and child process). Adjust the job size of the program. The graph we get reflects the growth of time. In this figure, we can see that in the 6 programs, time has a growth trend similar to a quadratic function with the increase of job size. This is in line with our previous analysis that the computational complexity of the three algorithms is quadratic.

Firstly, since all the programs take shorter time than the sequential program, it indecates that or program indeed has the performance improvement. At the same time, we found that mpi and mpi+openmp programs are shorter in time than sequential programs, indicating that our distributed design can accelerate computing efficiency the most. We found that when the process number is 2, distributed programs accelerate the program more than when the process number is 2. This observation is consistent with the computational complexity of distributed programs mentioned earlier, and we will analyze the specific multiples later.

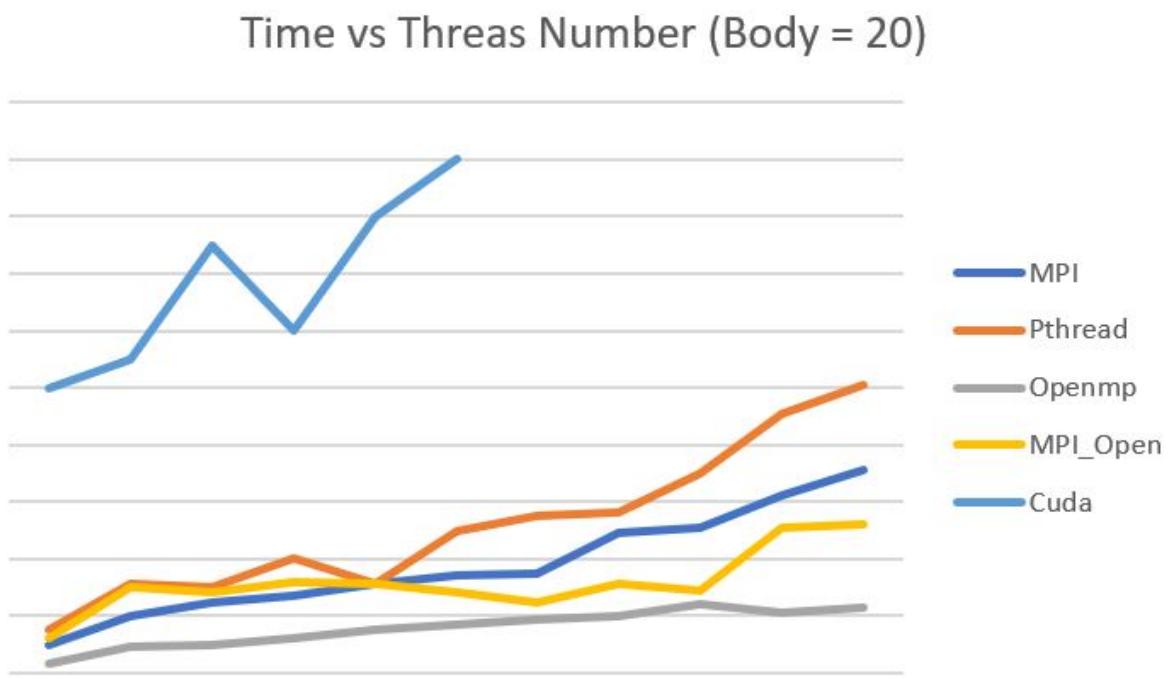

Figure 9: Comparison between all the algorithms: Time vs Procss numbers when bodies number is fixed to be 20

9.2 Comparison focusing on one algorithm

The above two pictures show the relationship between the time spent by 6 algorithms and the process number when the job size is fixed. For the fairness of comparison, this type of process number refers to the slave process of mpi and the child process of pthread.

The first figure 9 shows that when the body number is fixed at 20, how the time those algorithm will spend when the process numbers is increasing. However, it is anomalous that when the number of processors increases, the family uses them longer. This may be because, because the number of bodies is too small, each processor cannot allocate many bodies, so the locked time does not depend on the time used for calculation, but depends on the allocated processor, read Time spent writing and transferring memory. For example, the time used by cuda in this process is relatively old. The reason may be that I have a lot of Cudamecpy in my cuda program. This leads to a waste of time, so cuda will take a long time.

Now since the job size is much bigger, we can observe some useful information. We found that when the number of processes increases, the time has a slight downward trend as shown in figure 10. This trend is consistent with the computational complexity of the 5 previously analyzed. When the number of process numbers is larger, the amount of work allocated to each process will also become less, so the time taken to complete a single computing task will also become less. This effectively shows that distributed computing can reasonably increase the time used. And this tendency is more obvious when the body size is more than 1000, as shown in figure

At the same time, we also found that although the time used by mpi is very close to that of pthread, it still takes more time than pthread within the tolerance range. This point may be the same as the reason we analyzed in the previous section. Because in pthread, all child processes will be rewritten directly on the canvas, they do not need a

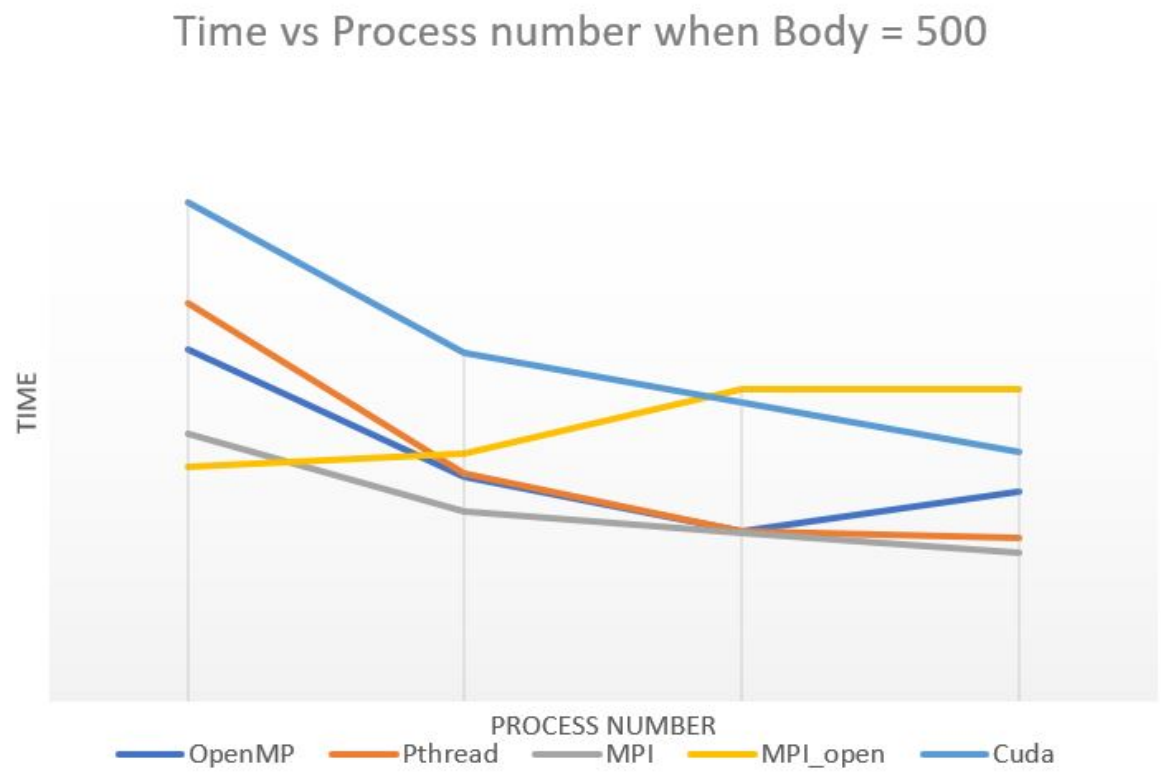

Figure 10: Comparison between all the algorithms: Time vs Process numbers when bodies number is fixed to be 500

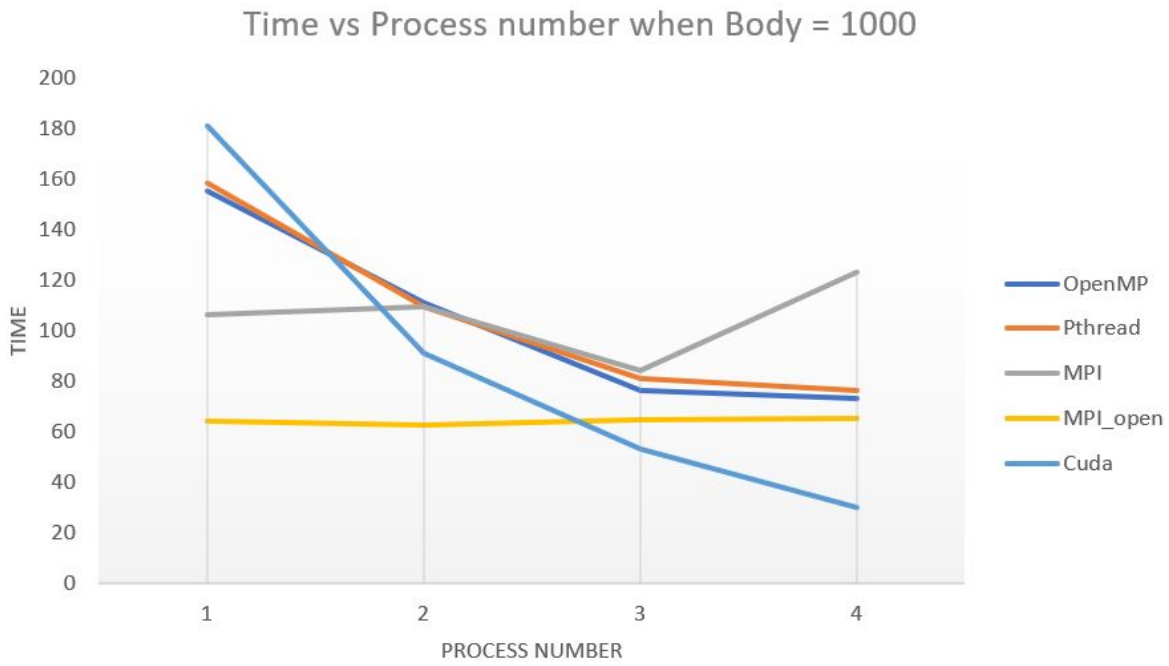

Figure 11: Comparison between all the algorithms: Time vs Process numbers when bodies number is fixed to be 1000

local buffer to store data. But in my mpi algorithm, I use the local buffer to store the data of the slave processes and pass this data to the MASTER process. At the same time, the MATER process also needs to write the received signal into the canvas. This process is very Time-consuming.

Moreover, I also observed that cuda has a very outstanding performance when body size is increasing. Although its performance may not be super good at the beginning when the processor number is small, but it is super cool when the processor number is increasing.

10 Conclusion and Future work

This focuses on the subject that how to write static scheduling program and write a pthread program and most importantly, OpenMP program and integrate the OpenMP with the MPI method. In this project, the N-body Simulation image is displayed by using Sequential, MPI, OpenMP, Pthread, MPI-OpenMP Hybrid methods, and cuda implementation. The above performance analysis clarifies why the MPI method will show a downward trend of performance and both of the Pthread and OpenMP have better performance than MPI when the body size is small and medium. When the body size grows, both of the MPI method and Pthread method shows a general downward trend of running time. The MPI method has a improvement in performance. And cuda performe the best when the job size is large and the device number increases.

Moreover, I also find that in the same condition, my implimentation of pthread is actually much faster than MPI. This is due to the face that in my practice, pthread directly write to the buffer which store the information of generated image while in MPI I need to use local buffer and then send the information to the master process, then the master process writes to the image buffer. This is actually time consuming point. And this point can be further improved.

Moreover, due to the resource limitation, some subtle and comprehensive experiment should be done in the further situation.