Assignment 3 N-body Simulation
CSC4005 Distributed and Parallel Computing

Cui Yuncong
School of Data Science
The Chinese University of Hong Kong, Shenzhen
118010045@link.cuhk.edu.cn

1. Introduction

The project is designed to simulate the N-body problem, which determines the effects of forces between bodies such as astronomical bodies attracted to each other through gravitational forces. The bodies are initially at rest. Their initial positions and masses are to be selected randomly (using a random number generator). The objective is to find the positions and movements of bodies in space that are subject to gravitational forces from other bodies.

The gravity between N-body should be described by the following equation

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

G is the gravitational constant. r is the distance between the bodies. m_1 and m_2 represent two entities masses.

Subject to a body will accelerate according to Newton's second law:

$$F = ma \tag{2}$$

2. Method

The below figures illustrate the output of the design of the Sequential method, MPI method, the Pthread method, OpenMP method and MPI-OpenMP Method.

2.1. Sequential Method

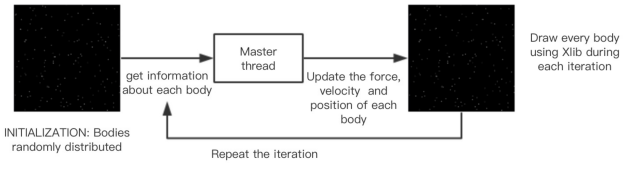

Figure 1: The Program Flow of Sequential N-body Simulation

2.2. MPI Method

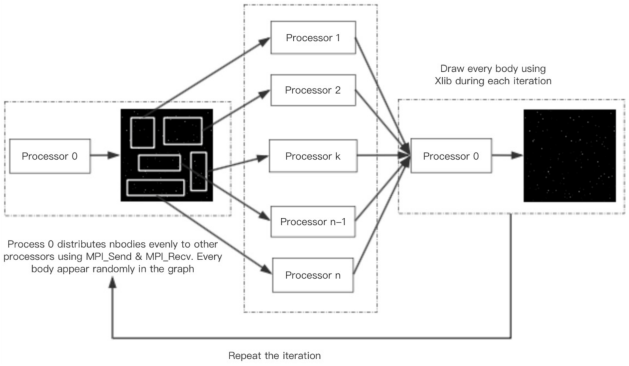

Figure 2: The Program Flow of MPI N-body Simulation

2.3. Pthread Method

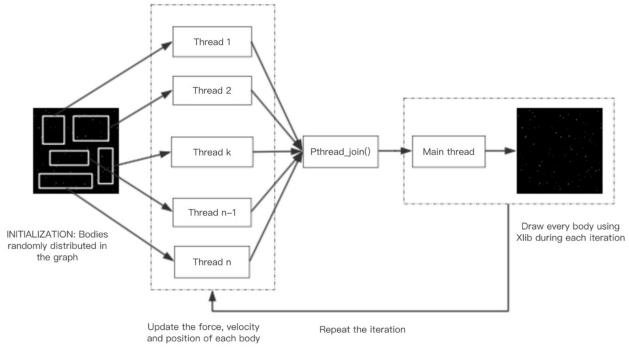

Figure 3: The Program Flow of Pthread N-body Simulation

2.4. OpenMP Method

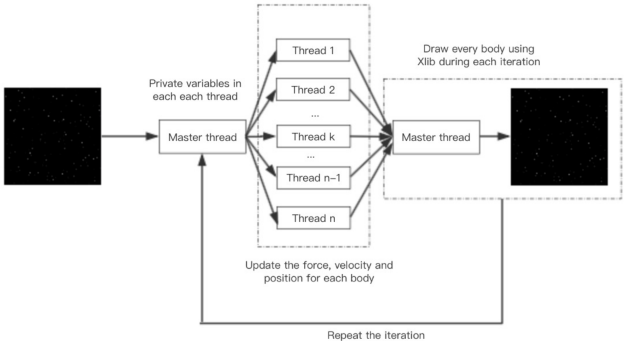

Figure 4: The Program Flow of OpenMP N-body Simulation

2.5. MPI-OpenMP Method

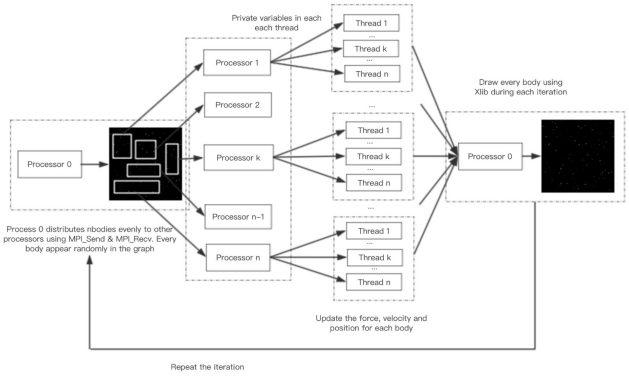

Figure 5: The Program Flow of MPI-OpenMP N-body Simulation

3. Experiment

3.1. Execution Steps

The default setting is not to show the result image. If the image needs to be printed, please change the bool variable PIC from false to true in each source code.

3.1.1 Sequential Method

Type “g++ hw3_seq.cpp -o seq.out -lX11” to compile the sequential method. Type “./seq.out” to execute the sequential method.

3.1.2 MPI Method

Type “mpic++ hw3_mpi.cpp -o mpi.out -lX11” to compile the pthread method. Type “mpirun -n 4 ./mpi.out” to execute the MPI method.

3.1.3 Pthread Method

Type “g++ hw3_pthread.cpp -o pthread.out -lX11 -lpthread” to compile the pthread method. Type “./pthread.out 4” to execute the Pthread method.

3.1.4 OpenMP Method

Type “g++ hw3_openmp.cpp -o openmp.out -lX11 -fopenmp” to compile the pthread method. Type “./openmp.out 4” to execute the OpenMP method.

3.1.5 MPI-OpenMP Method

Type “mpic++ hw3_mpi_openmp.cpp -o mpi_openmp.out -lX11 -fopenmp” to compile the pthread method. Type “mpirun -n 4 ./mpi_openmp.out 4” to execute the MPI-OpenMP method.

```
cuhksz_csc@CSC4005:~/Desktop/3$ g++ hw3_seq.cpp -o seq.out -lX11 && ./seq.out
Name: Cui Yuncong
Student ID: 118010045
Assignment 3, N-body simulation, Sequential Implementation
runTime is  13.024373
cuhksz_csc@CSC4005:~/Desktop/3$ mpic++ hw3_mpi.cpp -o mpi.out -lX11 && mpirun -n 4 ./mpi.out
Name: Cui Yuncong
Student ID: 118010045
Assignment 3, N-body simulation, MPI Implementation
runTime is  5.193481
cuhksz_csc@CSC4005:~/Desktop/3$ g++ hw3_pthread.cpp -o pthread.out -lX11 -lpthread && ./pthread.out 4
Name: Cui Yuncong
Student ID: 118010045
Assignment 3, N-body simulation, Pthread Implementation
runTime is  4.806012
cuhksz_csc@CSC4005:~/Desktop/3$ g++ hw3_openmp.cpp -o openmp.out -lX11 -fopenmp && ./openmp.out 4
Name: Cui Yuncong
Student ID: 118010045
Assignment 3, N-body simulation, OpenMP Implementation
runTime is  3.755611
cuhksz_csc@CSC4005:~/Desktop/3$ mpic++ hw3_mpi_openmp.cpp -o mpi_openmp.out -lX11 -fopenmp && mpirun -n 2 ./mpi_openmp.out 2
Name: Cui Yuncong
Student ID: 118010045
Assignment 3, N-body simulation, MPI-OpenMP Implementation
runTime is  5.335981
```

Figure 6: The Calculation Result Generated Methods

3.2. Data Analysis

Noticed that the display of the image is excluded from the final evaluation since the drawing process is extremely timeconsuming on the cluster server and time spent on the computation (or communication) is quite trivial compared with the running time of the drawing. In this case, we will not take the drawing time into consideration. The performance analysis is conducted in two dimensions, varying in body size and the number of processors / threads.

Figure 7-12 demonstrate the performance (running time) of the Sequential, MPI, OpenMp, Pthread method when the body size is 10, 50, 200, 400, 800 and 1000;

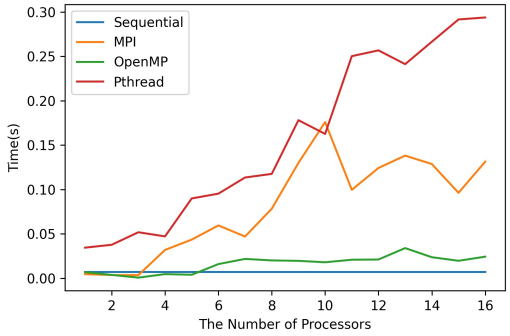

Figure 7: Performance Analysis of the MPI, Pthread, OpenMP and Sequential Method (10 Bodies)

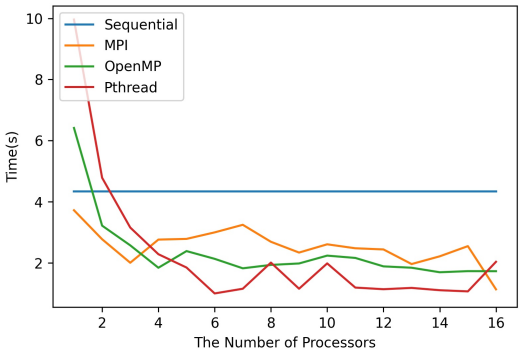

Figure 10: Performance Analysis of the MPI, Pthread, OpenMP and Sequential Method (400 Bodies)

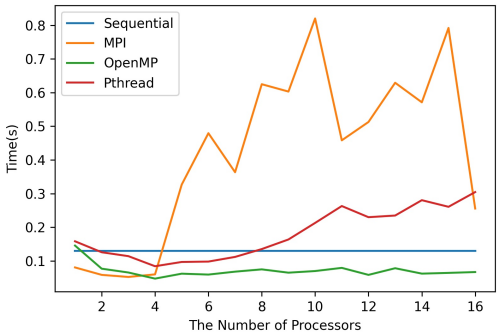

Figure 8: Performance Analysis of the MPI, Pthread, OpenMP and Sequential Method (50 Bodies)

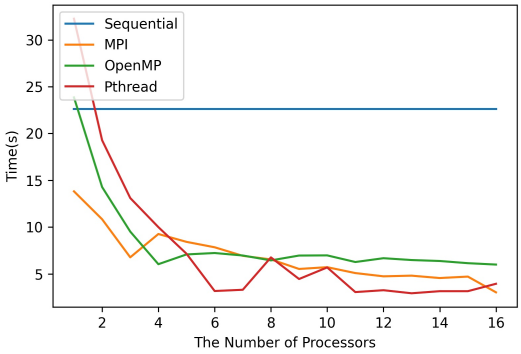

Figure 11: Performance Analysis of the MPI, Pthread, OpenMP and Sequential Method (800 Bodies)

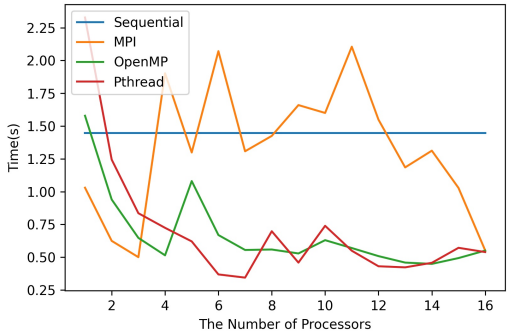

Figure 9: Performance Analysis of the MPI, Pthread, OpenMP and Sequential Method (200 Bodies)

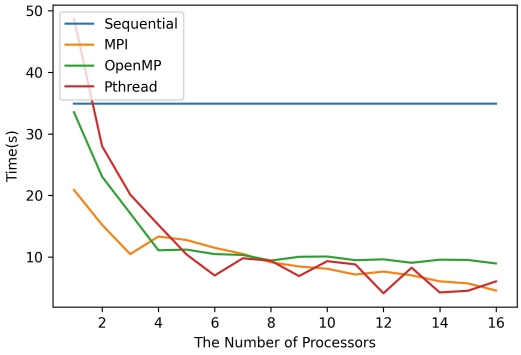

Figure 12: Performance Analysis of the MPI, Pthread, OpenMP and Sequential Method (1000 Bodies)

In general these results exhibit reasonable speedup as the number of participating processes increases. The estimated execution-time is $O(nm)$ for serial program. A parallel implementation can be estimated by $O(nm/p)$ where n and m are size and p is number of processes.

When the body size is small, say, 10, the performance of pthread method is worse than the MPI method. When the number of processors grows, the contrast is even distinct. The MPI method, in this case, have a worst performance when the number of processors grows from 4 to 10. It is because when the image size is small, the time spent on computation is trivial compares to the time spent on communication. As the number of processors grows, the overhead of the communication is even heavier. When the body size is medium and large(400 and 1000), the ratio of the computation time is larger, the computation time's increment surpasses the communication time's increment which leads to a performance gain of MPI compared to sequential method. when the body size 10, all of the three(MPI, Pthread, OpenMP) methods have a worse performance than sequential method. The reason why OpenMP and Pthread have a respectively better performance under small size of bodies is that they are based on the shared-memory system while MPI is based on the massage-passing system. The shared-memory system does not require that massage passes from one processors to another. They pass the massage by pointer which save a lot of time under small size.

When body size becomes medium(400), all of the MPI, Pthread and OpenMP have a better performance than Sequential method when the number of processors larger than 3. In this case, Pthread have the best performance over MPI and OpenMP method. OpenMp have a better performance than MPI. When the number of processors/ threads increases, all of them have a respective increase in performance.

4When body size becomes large(1000), we can see that the running time of MPI, OpenMP and Pthread show a general downward trend.The running time of all of them is quite smaller than the Sequential method. Its is because directly reading from the memory is faster than MPI_Send and MPI_Recv(). When the image size grows, the computation time surpasses the communication time, that is why the MPI have a better performance than previous experiment. But OpenMP still have the best performance.

In order to analyze the performance of the MPI+OpenMP. First, it needs to fix the numebr of processors of both the MPI version and the OpenMP version. The following figures show the running time of the MPI vs MPI-OpenMP versions and OpenMP vs MPI-OpenMP versions under different body size. The experiment, conducted the situation where the number of processors equals to 4. And body size are 10, 400 and 1000.

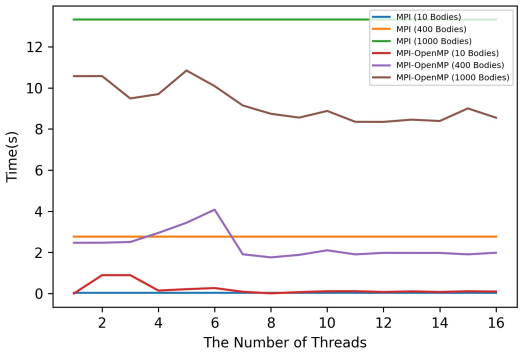

Figure 13: Performance Analysis of the MPI and MPI-OpenMP Method (Body Size: 10, 400 and 1000 with 4 Processors)

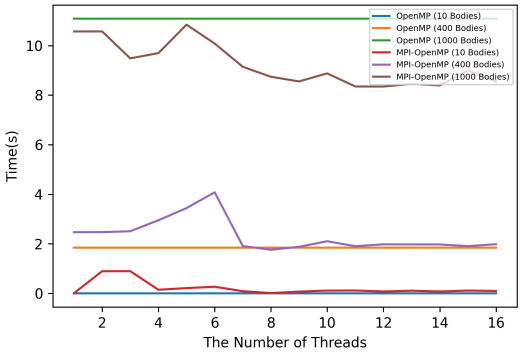

Figure 14: Performance Analysis of the OpenMP and MPI-OpenMP Method (Body Size: 10, 400 and 1000 with 4 Processors)

Figure 13 demonstrates the performance of the MPI and MPi-OpenMP programs when the body size is 10, 400 and 1000. When the body size is 10 and 400, as the number of thread grows from 1 to 6, the performance of hybrid programs are worse than only MPI program, but as the thread number keeps growing, the running time of the hybrid programs start to decrease but there is still not a huge gap between the MPI and the Hybrid. When the body size is large(1000), the performance of the Hybrid program will be worse as the numebr of thread increases.

Figure 14 demonstrates the performance of the OpenMP and MPI-OpenMP programs when the body size is 10, 400 and 1000. In this case, the MPI-OpenMP programs obviously conduct a better performance than the OpenMP prorgam.

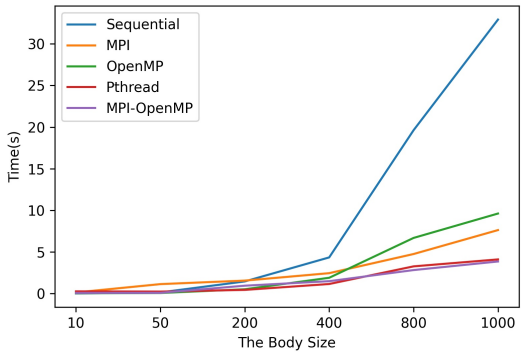

Figure 15: Performance Analysis of the Sequential, MPI, OpenMP, Pthread and MPI-OpenMP (12 Processors)

In Figure 15, the running time of MPI, OpenMP, Pthread, Sequential and MPI-OpenMP programs all grow when the image size grows. Noticed the value of x-coordinate is not proportionally increase. The running time of sequential method show like a polynomial curve, which is in proportion to the body size.

4. Code Implementation

This section only demonstrates the core code of in MPI since these parts of code are similar in each program.

This Figure 16 demonstrates the implementation of calculation the move of the body

```
void calculation(body_t *body, int body_num, int localStart, int localEnd)
{
    double x1, x2, y1, y2, m1, m2, xv1, xv2, yv1, yv2, r, f_x, f_y;

    for (int i = localStart; i < localEnd; i++)
    {
        body[i].x_f = 0;
        body[i].y_f = 0;
        for (int j = 0; j < body_num; j++)
        {
            x1 = body[i].x_pos;
            x2 = body[j].x_pos;
            y1 = body[i].y_pos;
            y2 = body[j].y_pos;
            m1 = body[i].mass;
            m2 = body[j].mass;
            xv1 = body[i].x_v;
            xv2 = body[j].x_v;
            yv1 = body[i].y_v;
            yv2 = body[j].y_v;

            if (x1 != x2 || y1 != y2)
            {
                r = sqrt(pow((x1 - x2), 2) + pow((y1 - y2), 2));
                f_x = G * m1 * m2 * (x2 - x1) / pow(r, 3);
                f_y = G * m1 * m2 * (y2 - y1) / pow(r, 3);

                if (r > 0.5)
                {
                    body[i].x_f += f_x;
                    body[i].y_f += f_y;
                }
                else
                {
                    body[i].x_v = ((m1 - m2) / (m1 + m2)) * xv1 + (2 * m2 / (m1 + m2)) * xv2;
                    body[j].x_v = (2 * m1 / (m1 + m2)) * xv1 + ((m2 - m1) / (m1 + m2)) * xv2;
                    body[i].y_v = ((m1 - m2) / (m1 + m2)) * yv1 + (2 * m2 / (m1 + m2)) * yv2;
                    body[j].y_v = (2 * m1 / (m1 + m2)) * yv1 + ((m2 - m1) / (m1 + m2)) * yv2;
                }
            }
        }
    }
}
```

Figure 16: Calculation the Move of the Body

This Figure 17 demonstrates the implementation of moving the body.

```
void move(body_t *body, int localStart, int localEnd)
{
    double m, x_a, y_a, x_v0, y_v0, x_d, y_d;

    for (int i = localStart; i < localEnd; i++)
    {
        m = body[i].mass;
        x_a = body[i].x_f / m;
        y_a = body[i].y_f / m;
        x_v0 = body[i].x_v;
        y_v0= body[i].y_v;

        x_d = x_v0 * delta_t + 0.5 * x_a * pow(delta_t, 2);
        y_d = y_v0 * delta_t + 0.5 * y_a * pow(delta_t, 2);

        body[i].x_pos += x_d;
        body[i].y_pos += y_d;

        if (body[i].x_pos < 0 || body[i].x_pos > Xborder)
            body[i].x_v = -body[i].x_v;

        if (body[i].y_pos < 0 || body[i].y_pos > Yborder)
            body[i].y_v = -body[i].y_v;
    }
}
```

Figure 17: Move the Body

This Figure 18 demonstrates the implementation of the collision.

```
void collision(body_t *body, int body_num, int localStart, int localEnd)
{
    double dx, dy, r, m1, m2, xv1, xv2, yv1, yv2;

    for (int i = localStart; i < localEnd; i++)
    {
        for(int j = 0; j < body_num; j++)
        {
            if (j != i)
            {
                dx = body[j].x_pos - body[i].x_pos;
                dy = body[j].y_pos - body[i].y_pos;
                r = sqrt(pow(dx, 2) + pow(dy, 2));
                if (r < 0.5)
                {
                    m1 = body[i].mass;
                    m2 = body[j].mass;
                    xv1 = body[i].x_v;
                    xv2 = body[j].x_v;
                    yv1 = body[i].y_v;
                    yv2 = body[j].y_v;
                    body[i].x_v = ((m1 - m2) / (m1 + m2)) * xv1 + (2 * m2 / (m1 + m2)) * xv2;
                    body[j].x_v = (2 * m1 / (m1 + m2)) * xv1 + ((m2 - m1) / (m1 + m2)) * xv2;
                    body[i].y_v = ((m1 - m2) / (m1 + m2)) * yv1 + (2 * m2 / (m1 + m2)) * yv2;
                    body[j].y_v = (2 * m1 / (m1 + m2)) * yv1 + ((m2 - m1) / (m1 + m2)) * yv2;
                }
            }
        }
        if (body[i].x_pos < minX)
        {
            if (body[i].x_v < 0)
                body[i].x_v = loss * body[i].x_v;
            body[i].x_pos = minX;
        }
        if (body[i].x_pos > maxX)
        {
            if (body[i].x_v > 0)
                body[i].x_v = loss * body[i].x_v;
            body[i].x_pos = maxX;
        }
        if (body[i].y_pos < minY)
        {
            if (body[i].y_v < 0)
                body[i].y_v = loss * body[i].y_v;
            body[i].y_pos = minY;
        }
        if (body[i].y_pos > maxY)
        {
            if (body[i].y_v > 0)
                body[i].y_v = loss * body[i].y_v;
            body[i].y_pos = maxY;
        }
    }
}
```

Figure 18: Deal with the Collision

5. Conclusion

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 and MPI-OpenMP Hybrid methods. 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(10) and medium(400). When the body size grows (1000), both of the MPI method and Pthread method shows a general downward trend of running time. The MPI method has a improvement in performance.