Project3: N-body Simulation

Name: Xinyu Xie

Student ID: 119020059

Introduction

N-body simulation is a simulation of a dynamical system of particles. It is usually under the influence of physical forces like gravity. N-body simulations are widely used tools in astrophysics. 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.

The initial positions and masses of N bodies are selected randomly using a random number generator. The gravity between N-body is calculated using the equation: $F = \frac{Gm_1m_2}{r^2}$. The collision and bouncing between bodies and between body and walls are also considered for updating the positions.

In this project, I implemented the seqential version, the parallel MPI version, the parallel Pthread version, the parallel OpenMP version, the parallel CUDA version, the parallel MPI + OpenMP version, of the N-body simulation. The detailed implementation methods, results and conclusions are provided in the following sections.

Method

Sequential N-body Simulation

The following are the steps to perform sequential N-body simulation:

1. Generate data for each body by randomly assigning their masses, positions, and velocities.
2. In every iteration, for each body, calulate each force between it and all other bodies using the formula $F = \frac{Gm_1m_2}{r^2}$. The force is decomposed into x and y directions. Add all forces between each body and all other bodies to get the total force on each body. Calculate the aceleration in x and y directions using the formula $a = \frac{F}{m}$. Update velocity using the formula $v = v + at$.
3. In every iteration, for each body, update its position using the new updated velocity. Check the collision between two objects and between object and wall. If two objects collide, the velocity of these two objects should be

reversed. If the object collides with wall, either x direction or y direction velocity of the object should be reversed.

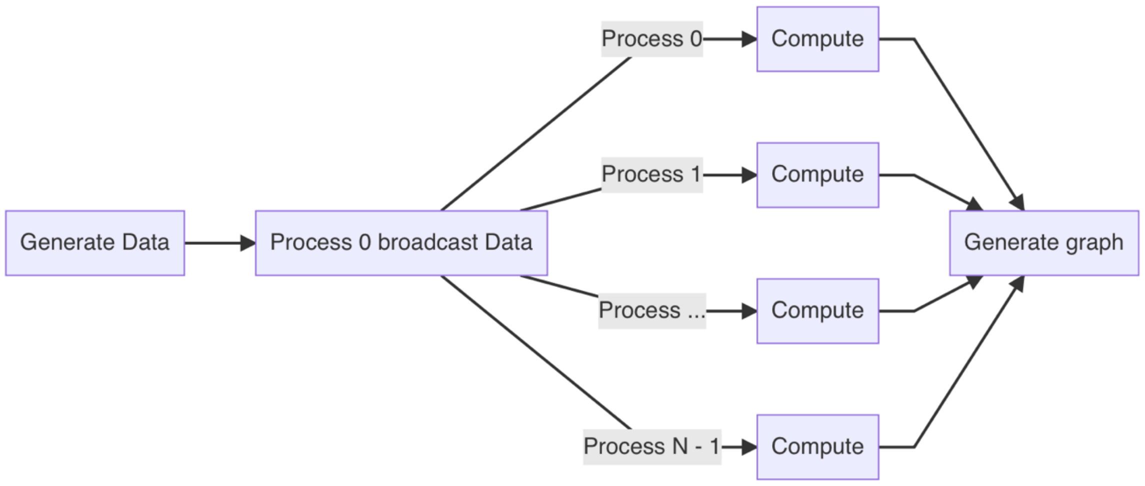
4. Draw the corresponding graph.

Parallel N-body Simulation using MPI

The following are the steps to perform parallel N-body simulation using MPI:

5. Generate data for each body by randomly assigning their masses, positions, and velocities in master process.
6. In total, N bodies are equally distributed to n processes, respectively. Master process distribute equal number of bodies to each slave process.
7. Inside each slave process, for each body in local data, update its corresponding velocity and position given in the sequential method. Each slave process broadcasts its data vx and vy after updating velocity and broadcasts x, y, vx, vy after updating positions to all other slave processes. Wait for all slave processes to reach barrier to continue next time iteration.
8. Master process collect the result from each slave process and then draw the corresponding graph.

This flow chart is illustrated in the following figure:



Parallel N-body Simulation using Pthread

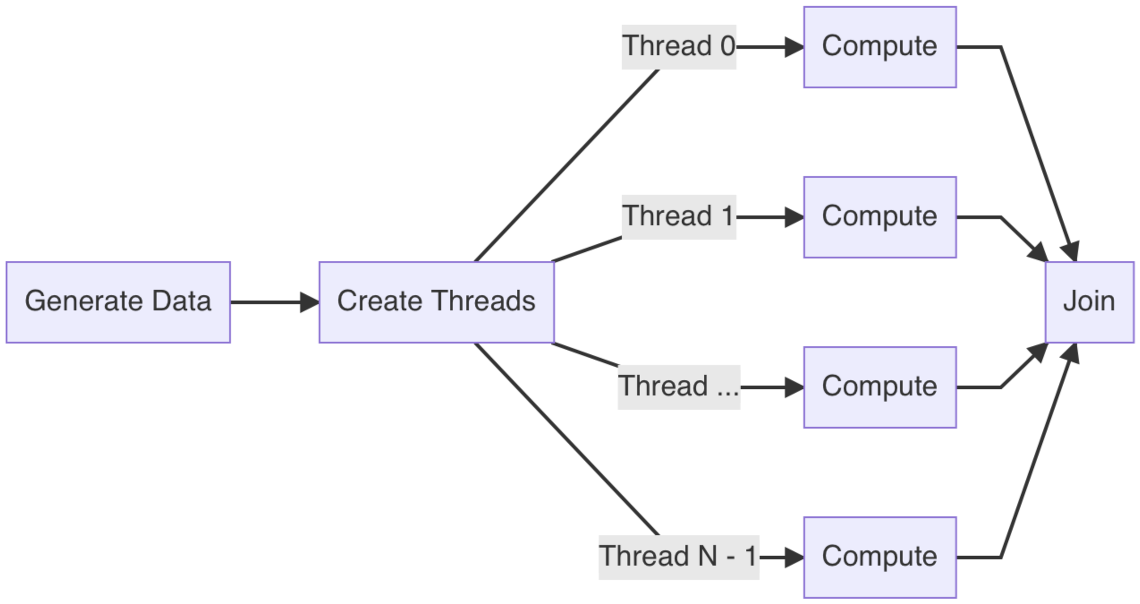
The following are the steps to perform parallel N-body simulation using Pthread:

1. Generate data for each body by randomly assigning their masses, positions, and velocities in master process.
2. In total, N bodies are equally partitioned into n parts.
3. Create n child threads.
4. Inside each child thread, for each body in local data, update its corresponding velocity and position given in the sequential method. When an object in the thread collide with other object, use a mutex lock to lock the velocity of the

other object before modification. Wait for all threads to reach barrier to continue next time iteration.

5. Each child thread will join together.

This flow chart is illustrated in the following figure:

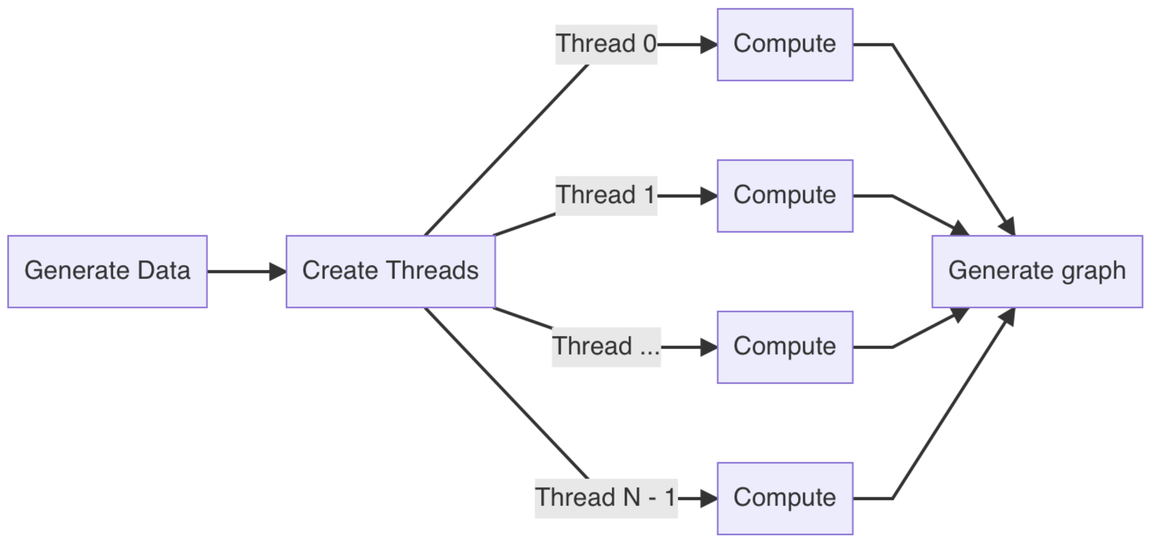


Parallel N-body Simulation using OpenMP

The following are the steps to perform parallel N-body simulation using OpenMP:

1. Generate data for each body by randomly assigning their masses, positions, and velocities in master process.
2. In every iteration, for each body, using parallely for to calulate each force between it and all other bodies. Create critical session when updating the velocity of each body.
3. In every iteration, for each body, using parallely for to update its position using the new updated velocity. Check the collision between two objects. Create critical session when updating the velocity of each body.
4. Draw the corresponding graph.

This flow chart is illustrated in the following figure:

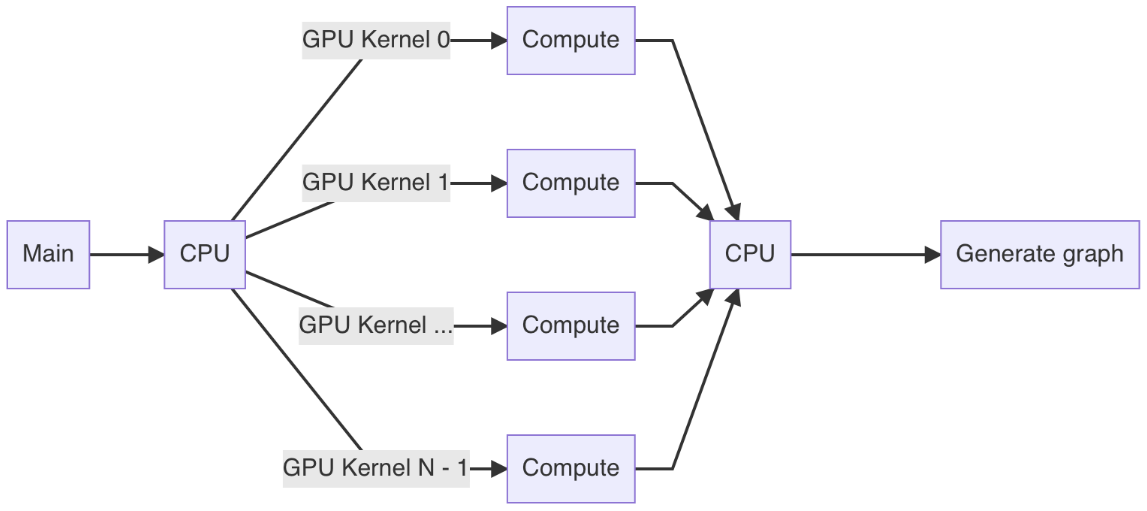


Parallel N-body Simulation using CUDA

The following are the steps to perform parallel N-body simulation using CUDA:

1. Generate data for each body by randomly assigning their masses, positions, and velocities in master function.
2. The main function fork n threads. In total, N bodies are equally partitioned into n parts, each thread will have 1 or 0 number of body.
3. Inside each thread having 1 body, update its corresponding velocity and position given in the sequential method.
4. Draw the corresponding graph.

This flow chart is illustrated in the following figure:

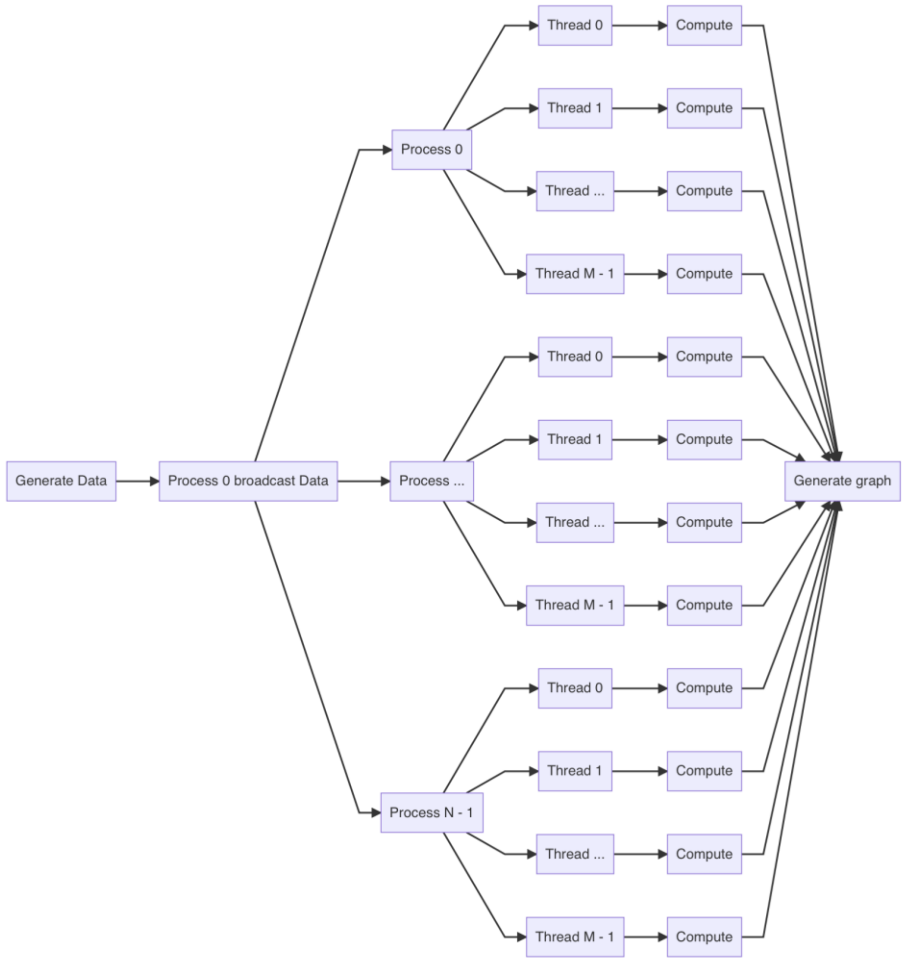


Bonus: Parallel N-body Simulation using MPI + OpenMP

The following are the steps to perform parallel N-body simulation using MPI + OpenMP:

1. Generate data for each body by randomly assigning their masses, positions, and velocities in master process.
2. In total, N bodies are equally distributed to n processes, respectively. Master process distribute equal number of bodies to each slave process.
3. Inside each slave process, use OpenMP to generate more threads. In every iteration, for each body in local data, using parallel for to update its corresponding velocity and position given in the sequential method. Each slave process broadcasts its data vx and vy after updating velocity and broadcast sx, y, vx, vy after updating positions to all other slave processes. Wait for all slave processes to reach barrier to continue next time iteration.
4. Master process collect the result from each slave process and then draw the corresponding graph.

This flow chart is illustrated in the following figure:



Implementation

Sequential N-body Simulation

The implementation of sequential N-body simulation directly follows the method mentioned above.

The following is the core part of detailed implementation code:

```
void update_position() {
    for (int i = 0; i < n_body; ++i) {
        x[i] += vx[i] * dt;
        y[i] += vy[i] * dt;

        // check collision between two objects
        for (int j = 0; j < n_body; ++j) {
            if (i == j) continue;
            double distance_sqrt = get_distance(i, j, x, y);
            if (distance_sqrt * distance_sqrt <= 4 * radius2) {
                vx[i] = -vx[i];
                vy[i] = -vy[i];
                vx[j] = -vx[j];
                vy[j] = -vy[j];
            }
        }

        // check collision between object and wall
        if (x[i] <= 0 || x[i] >= bound_x) {
            vx[i] = -vx[i];
        }
        if (y[i] <= 0 || y[i] >= bound_y) {
            vy[i] = -vy[i];
        }
    }
}

void update_velocity() {
    for (int i = 0; i < n_body; i++) {
        double fx = 0;
        double fy = 0;
        // calculate force from all other objects
        for (int j = 0; j < n_body; ++j) {
            if (i == j) continue;
            double distance_sqrt = get_distance(i, j, x, y);
            double f = gravity_const * m[i] * m[j] / (distance_sqrt * d
istance_sqrt + err);
            fx += f * (x[j] - x[i]) / distance_sqrt;
            fy += f * (y[j] - y[i]) / distance_sqrt;
        }
        // update velocity
        double ax = fx / m[i];
        double ay = fy / m[i];
        vx[i] += ax * dt;
        vy[i] += ay * dt;
    }
}
```

Parallel N-body Simulation using MPI

The implementation of parallel N-body simulation using MPI is divided into following steps:
1. Do initialization and distribution of N bodies. 2. Master process distributes data points to each slave process using MPI_Bcast. 3. Perform local computation in each slave process.
4. Master process gather results and generate results.

The following is the core part of detailed implementation code:

```
void update_once() {
    update_velocity();
    for (int i = 0; i < world_size; ++i) {
        MPI_Bcast(&vx[displacement[i]], send_counts[i], MPI_DOUBLE, i, 
MPI_COMM_WORLD);
        MPI_Bcast(&vy[displacement[i]], send_counts[i], MPI_DOUBLE, i, 
MPI_COMM_WORLD);
        MPI_Barrier(MPI_COMM_WORLD);
    }
    update_position();
    for (int i = 0; i < world_size; ++i) {
        MPI_Bcast(&vx[displacement[i]], send_counts[i], MPI_DOUBLE, i, 
MPI_COMM_WORLD);
        MPI_Bcast(&vy[displacement[i]], send_counts[i], MPI_DOUBLE, i, 
MPI_COMM_WORLD);
        MPI_Bcast(&x[displacement[i]], send_counts[i], MPI_DOUBLE, i, M
PI_COMM_WORLD);
        MPI_Bcast(&y[displacement[i]], send_counts[i], MPI_DOUBLE, i, M
PI_COMM_WORLD);
        MPI_Barrier(MPI_COMM_WORLD);
    }
}

void slave() {
    MPI_Bcast(m, n_body, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(x, n_body, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(y, n_body, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(vx, n_body, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(vy, n_body, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    for (int i = 0; i < n_iteration; i++) {
        update_once();
    }
}

void master() {
    generate_data(m, x, y, vx, vy, n_body);
    MPI_Bcast(m, n_body, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(x, n_body, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(y, n_body, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(vx, n_body, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(vy, n_body, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    for (int i = 0; i < n_iteration; i++) {
        t1 = std::chrono::high_resolution_clock::now();
        update_once();
        t2 = std::chrono::high_resolution_clock::now();
        time_span = t2 - t1;
        printf("Iteration %d, elapsed time: %.3f\n", i, time_span);
    }
}
```

Parallel N-body Simulation using Pthread

The implementation of parallel N-body simulation using Pthread is divided into following steps:

1. Do partition of N bodies.
2. Create a thread pool containing n child threads using pthread_create.
3. Perform local computation in each child thread.
4. Each child thread join together using pthread_join.

The following is the core part of detailed implementation code:

```
typedef struct {
    int start; // array start position of whole data array
    int size; // size of array
    int n_iterations;
} Args;

void* worker(void* args) {
    Args* my_arg = (Args*) args;
    int start = my_arg->start;
    int size = my_arg->size;
    int n_iterations = my_arg->n_iterations;

    for (int i = 0; i < n_iterations; i++) {
        update_velocity(start, size);
        update_position(start, size);
        pthread_barrier_wait(&barrier);
    }
    pthread_exit(NULL);
    return nullptr;
}

void master() {
    m = new double[n_body];
    x = new double[n_body];
    y = new double[n_body];
    vx = new double[n_body];
    vy = new double[n_body];
    generate_data(m, x, y, vx, vy, n_body);

    Args args[n_thd]; // arguments for all threads
    int size = n_body / n_thd;
    int remainder = n_body % n_thd;
    // allocate data array for each thread
    for (int thd = 0; thd < remainder; thd++) {
        args[thd].start = thd * (size + 1);
        args[thd].size = size + 1;
        args[thd].n_iterations = n_iteration;
    }
    for (int thd = remainder; thd < n_thd; thd++) {
        args[thd].start = thd * size + remainder;
        args[thd].size = size;
        args[thd].n_iterations = n_iteration;
    }

    pthread_mutex_init(&mutex, NULL);
    pthread_barrier_init(&barrier, NULL, n_thd + 1);
    pthread_t thds[n_thd]; // thread pool
    for (int thd = 0; thd < n_thd; thd++) pthread_create(&thds[thd], NU
LL, worker, &args[thd]);

    for (int i = 0; i < n_iteration; i++) {
        t1 = std::chrono::high_resolution_clock::now();
        pthread_barrier_wait(&barrier);
        t2 = std::chrono::high_resolution_clock::now();
        time_span = t2 - t1;
        printf("Iteration %d, elapsed time: %.3f\n", i, time_span);
    }

    for (int thd = 0; thd < n_thd; thd++) pthread_join(thds[thd], NULL);
    delete[] m;
    delete[] x;
    delete[] y;
    delete[] vx;
    delete[] vy;
    pthread_barrier_destroy(&barrier);
    pthread_mutex_destroy(&mutex);
}
```

Parallel N-body Simulation using OpenMP

The implementation of parallel N-body simulation using OpenMP is divided into following steps:

1. Do initialization of N bodies.
2. Using parallel for for each thread to perform local computation.
3. Collect final results.

The following is the core part of detailed implementation code:

```
void update_position(double* m, double* x, double* y, double* vx, doubl
e* vy, int n) {
    #pragma omp parallel for
    for (int i = 0; i < n; ++i) {
        x[i] += vx[i] * dt;
        y[i] += vy[i] * dt;

        // check collision between two objects
        for (int j = 0; j < n; ++j) {
            if (i == j) continue;
            double distance_sqrt = get_distance(i, j, x, y);
            if (distance_sqrt * distance_sqrt <= 4 * radius2) {
                #pragma omp critical
                {
                    vx[i] = -vx[i];
                    vy[i] = -vy[i];
                    vx[j] = -vx[j];
                    vy[j] = -vy[j];
                }
            }
        }

        // check collision between object and wall
        if (x[i] <= 0 || x[i] >= bound_x) {
            #pragma omp critical
            {
                vx[i] = -vx[i];
            }
        }
        if (y[i] <= 0 || y[i] >= bound_y) {
            #pragma omp critical
            {
                vy[i] = -vy[i];
            }
        }
    }
}

void update_velocity(double* m, double* x, double* y, double* vx, doubl
e* vy, int n) {
    #pragma omp parallel for
    for (int i = 0; i < n; i++) {
        double fx = 0;
        double fy = 0;
        // calculate force from all other objects
        for (int j = 0; j < n; ++j) {
            if (i == j) continue;
            double distance_sqrt = get_distance(i, j, x, y);
            double f = gravity_const * m[i] * m[j] / (distance_sqrt * d
istance_sqrt + err);
            fx += f * (x[j] - x[i]) / distance_sqrt;
            fy += f * (y[j] - y[i]) / distance_sqrt;
        }
        // update velocity
        double ax = fx / m[i];
        double ay = fy / m[i];
        #pragma omp critical
        {
            vx[i] += ax * dt;
            vy[i] += ay * dt;
        }
    }
}

void master() {
    double* m = new double[n_body];
    double* x = new double[n_body];
    double* y = new double[n_body];
    double* vx = new double[n_body];
    double* vy = new double[n_body];
    generate_data(m, x, y, vx, vy, n_body);

    for (int i = 0; i < n_iteration; i++){
        t1 = std::chrono::high_resolution_clock::now();
        omp_set_num_threads(n_omp_threads);
        update_velocity(m, x, y, vx, vy, n_body);
        update_position(m, x, y, vx, vy, n_body);
        t2 = std::chrono::high_resolution_clock::now();
        time_span = t2 - t1;
        printf("Iteration %d, elapsed time: %.3f\n", i, time_span);
    }
    delete[] m;
    delete[] x;
    delete[] y;
    delete[] vx;
    delete[] vy;
}
```

Parallel N-body Simulation using CUDA

The implementation of parallel N-body simulation using CUDA is divided into following steps:

1. Do initialization of N bodies.
2. Create a thread pool containing n child threads and do partition of N bodies.
3. Perform local computation in each thread.
4. Each thread join together.

The following is the core part of detailed implementation code:

```
__global__ void update_position(double *x, double *y, double *vx, doubl
e *vy, int n) {
    int i_thd = blockDim.x * blockIdx.x + threadIdx.x;
    if (i_thd < n) {
        int i = i_thd;
        x[i] += vx[i] * dt;
        y[i] += vy[i] * dt;

        // check collision between two objects
        for (int j = 0; j < n; ++j) {
            if (i == j) continue;
            double distance_sqrt = sqrt((x[i] - x[j]) * (x[i] - x[j]) +
 (y[i] - y[j]) * (y[i] - y[j]));
            if (distance_sqrt * distance_sqrt <= 4 * radius2) {
                vx[i] = -vx[i];
                vy[i] = -vy[i];
                vx[j] = -vx[j];
                vy[j] = -vy[j];
            }
        }

        // check collision between object and wall
        if (x[i] <= 0 || x[i] >= bound_x) {
            vx[i] = -vx[i];
        }
        if (y[i] <= 0 || y[i] >= bound_y) {
            vy[i] = -vy[i];
        }
    }
}

__global__ void update_velocity(double *m, double *x, double *y, double
 *vx, double *vy, int n) {
    int i_thd = blockDim.x * blockIdx.x + threadIdx.x;
    if (i_thd < n) {
        int i = i_thd;
        double fx = 0;
        double fy = 0;
        // calculate force from all other objects
        for (int j = 0; j < n; ++j) {
            if (i == j) continue;
            double distance_sqrt = sqrt((x[i] - x[j]) * (x[i] - x[j]) +
 (y[i] - y[j]) * (y[i] - y[j]));
            double f = gravity_const * m[i] * m[j] / (distance_sqrt * d
istance_sqrt + err);
            fx += f * (x[j] - x[i]) / distance_sqrt;
            fy += f * (y[j] - y[i]) / distance_sqrt;
        }
        // update velocity
        double ax = fx / m[i];
        double ay = fy / m[i];
        vx[i] += ax * dt;
        vy[i] += ay * dt;
    }
}

void master() {
    m = new double[n_body];
    x = new double[n_body];
    y = new double[n_body];
    vx = new double[n_body];
    vy = new double[n_body];
    generate_data(m, x, y, vx, vy, n_body);

    double *device_m;
    double *device_x;
    double *device_y;
    double *device_vx;
    double *device_vy;

    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), cudaMemcpyHostToDe
vice);
    cudaMemcpy(device_x, x, n_body * sizeof(double), cudaMemcpyHostToDe
vice);
    cudaMemcpy(device_y, y, n_body * sizeof(double), cudaMemcpyHostToDe
vice);
    cudaMemcpy(device_vx, vx, n_body * sizeof(double), cudaMemcpyHostTo
Device);
    cudaMemcpy(device_vy, vy, n_body * sizeof(double), cudaMemcpyHostTo
Device);
```

```
    int n_block = n_body / block_size + 1;

    for (int i = 0; i < n_iteration; i++){
        t1 = std::chrono::high_resolution_clock::now();
        update_velocity<<<n_block, block_size>>>(device_m, device_x, de
vice_y, device_vx, device_vy, n_body);
        update_position<<<n_block, block_size>>>(device_x, device_y, de
vice_vx, device_vy, n_body);

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

        t2 = std::chrono::high_resolution_clock::now();
        time_span = t2 - t1;
        printf("Iteration %d, elapsed time: %.3f\n", i, time_span);
    }

    cudaFree(device_m);
    cudaFree(device_x);
    cudaFree(device_y);
    cudaFree(device_vx);
    cudaFree(device_vy);

    delete[] m;
    delete[] x;
    delete[] y;
    delete[] vx;
    delete[] vy;
}
```

Bonus: Parallel N-body Simulation using MPI + OpenMP

The implementation of parallel N-body simulation using MPI + OpenMP is divided into following steps:

1. Do initialization and distribution of N bodies.
2. Master process distributes data points to each slave process using MPI_Bcast.
3. Perform local computation in each slave process using parallel for for each thread.
4. Master process gather results and generate results.

The following is the core part of detailed implementation code:

```
void update_position() {
    #pragma omp parallel for
    for (int i = displacement[my_rank]; i < displacement[my_rank + 1];
++i) {
        x[i] += vx[i] * dt;
        y[i] += vy[i] * dt;

        // check collision between two objects
        for (int j = 0; j < n_body; ++j) {
            if (i == j) continue;
            double distance_sqrt = get_distance(i, j, x, y);
            if (distance_sqrt * distance_sqrt <= 4 * radius2) {
                #pragma omp critical
                {
                    vx[i] = -vx[i];
                    vy[i] = -vy[i];
                    vx[j] = -vx[j];
                    vy[j] = -vy[j];
                }
            }
        }

        // check collision between object and wall
        if (x[i] <= 0 || x[i] >= bound_x) {
            #pragma omp critical
            {
                vx[i] = -vx[i];
            }
        }
        if (y[i] <= 0 || y[i] >= bound_y) {
            #pragma omp critical
            {
                vy[i] = -vy[i];
            }
        }
    }
}

void update_velocity() {
    #pragma omp parallel for
    for (int i = displacement[my_rank]; i < displacement[my_rank + 1];
i++) {
        double fx = 0;
        double fy = 0;
        // calculate force from all other objects
        for (int j = 0; j < n_body; ++j) {
            if (i == j) continue;
            double distance_sqrt = get_distance(i, j, x, y);
            double f = gravity_const * m[i] * m[j] / (distance_sqrt * d
istance_sqrt + err);
            fx += f * (x[j] - x[i]) / distance_sqrt;
            fy += f * (y[j] - y[i]) / distance_sqrt;
        }
        // update velocity
        double ax = fx / m[i];
        double ay = fy / m[i];
        #pragma omp critical
        {
            vx[i] += ax * dt;
            vy[i] += ay * dt;
        }
    }
}

void update_once() {
    omp_set_num_threads(n_omp_threads);
    update_velocity();
    for (int i = 0; i < world_size; ++i) {
        MPI_Bcast(&vx[displacement[i]], send_counts[i], MPI_DOUBLE, i,
MPI_COMM_WORLD);
        MPI_Bcast(&vy[displacement[i]], send_counts[i], MPI_DOUBLE, i,
MPI_COMM_WORLD);
        MPI_Barrier(MPI_COMM_WORLD);
    }
    update_position();
    for (int i = 0; i < world_size; ++i) {
        MPI_Bcast(&vx[displacement[i]], send_counts[i], MPI_DOUBLE, i,
MPI_COMM_WORLD);
        MPI_Bcast(&vy[displacement[i]], send_counts[i], MPI_DOUBLE, i,
MPI_COMM_WORLD);
        MPI_Bcast(&x[displacement[i]], send_counts[i], MPI_DOUBLE, i, M
PI_COMM_WORLD);
        MPI_Bcast(&y[displacement[i]], send_counts[i], MPI_DOUBLE, i, M
PI_COMM_WORLD);
        MPI_Barrier(MPI_COMM_WORLD);
    }
}

void slave() {
    MPI_Bcast(m, n_body, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(x, n_body, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(y, n_body, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(vx, n_body, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(vy, n_body, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    for (int i = 0; i < n_iteration; i++) {
        update_once();
    }
}


void master() {
    generate_data(m, x, y, vx, vy, n_body);
    MPI_Bcast(m, n_body, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(x, n_body, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(y, n_body, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(vx, n_body, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Bcast(vy, n_body, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    for (int i = 0; i < n_iteration; i++) {
        t1 = std::chrono::high_resolution_clock::now();
        update_once();
        t2 = std::chrono::high_resolution_clock::now();
        time_span = t2 - t1;
        printf("Iteration %d, elapsed time: %.3f\n", i, time_span);
    }
}
```

Execution

The following are the steps for execution:

1. To build, go to can root directory, run the following shell scripts to build:

   scripts/cmake_release.sh for non-GUI version

   scripts/cmake_release_gui.sh for GUI version

2. To run, go to the build directory:
3. Use "./seq [n body] [n iteration]" to run sequential version without GUI.
4. Use "./seqg [n body] [n iteration]" to run sequential version with GUI.
5. Use "mpirun -np [number of processes] ./mpi [n body] [n iteration]" to run parallel version using MPI without GUI.
6. Use "mpirun -np [number of processes] ./mpig [n body] [n iteration]" to run parallel version using MPI with GUI.
7. Use "./pthread [n body] [n iteration] [number of threads]" to run parallel version using Pthread without GUI.
8. Use "./pthreadg [n body] [n iteration] [number of threads]" to run parallel version using Pthread with GUI.
9. Use "./cuda [n body] [n iteration]" to run parallel version using CUDA without GUI.
10. Use "./cudag [n body] [n iteration]" to run parallel version using CUDA with GUI.

11. Use “openmp [n body] [n iteration] [number of omp threads]” to run parallel version using OpenMP without GUI.

12. Use “openmpg [n body] [n iteration] [number of omp threads]” to run parallel version using OpenMP with GUI.

13. Use “mpirun -np [number of processes] ./mpi [n body] [n iteration]” to run parallel version using MPI + OpenMP without GUI.

14. Use “mpirun -np [number of processes] ./mpig [n body] [n iteration]” to run parallel version using MPI + OpenMP with GUI.

The following are the steps to do experiments in hpc:

1. Go to project3 folder, compile all files using “make all”.

2. Config the task template batch file by specifying job name, nodes number, process number, number of CPU cores per process, total memory limit, time limit, and partition name shown as the following example.

   #!/bin/bash

   #SBATCH –job-name=119020059_seq_{size} # Job name

   #SBATCH –nodes=1 # Run all processes on a single node

   #SBATCH –ntasks=1 # number of processes = 4

   #SBATCH –cpus-per-task=1 # Number of CPU cores per process

   #SBATCH –mem=1000mb # Total memory limit

   #SBATCH –time=00:60:00 # Time limit hrs:min:sec

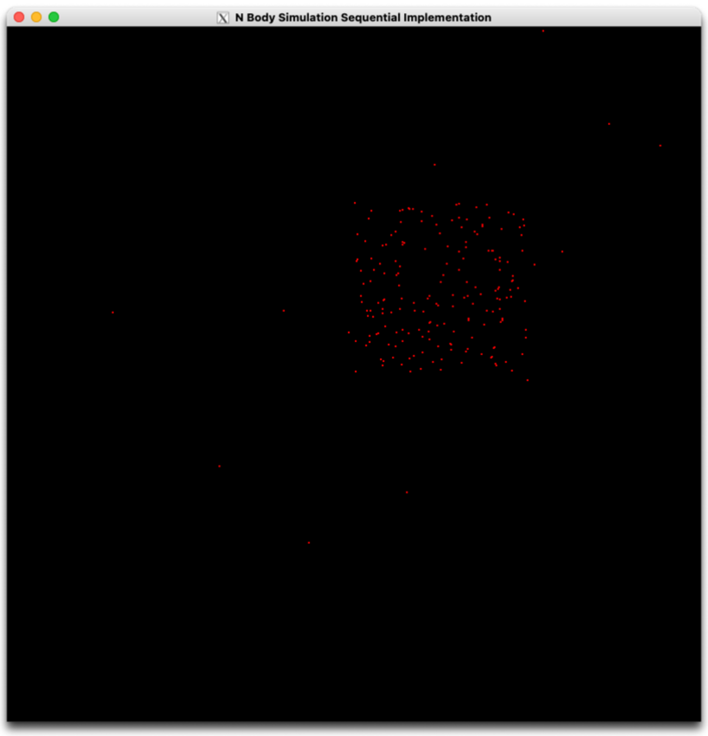
   #SBATCH –partition=Debug # Partition name: Project or Debug (Debug is default)

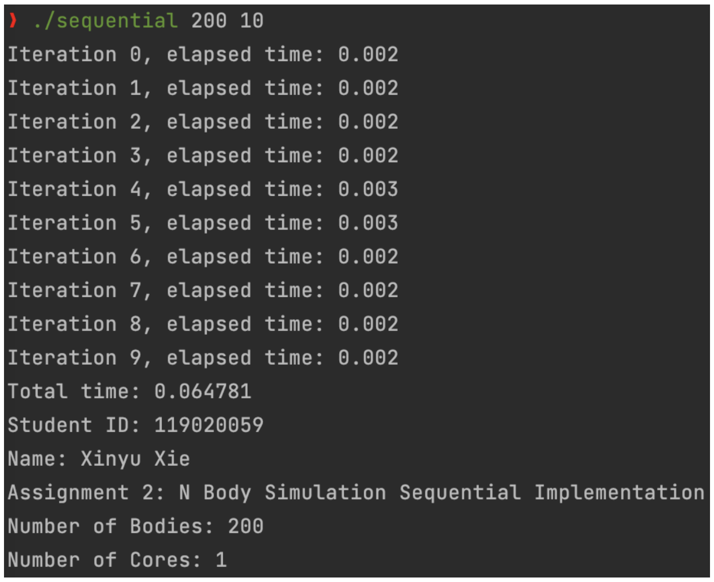
3. SSH to hpc and run “sbatch [batch file path]” to submit the job.

4. Collect running output file “slurm-[num].out”.

Sample Outputs

Image size = 800 × 800, Number of bodies = 200, Number of iterations = 10000. The following is the output:





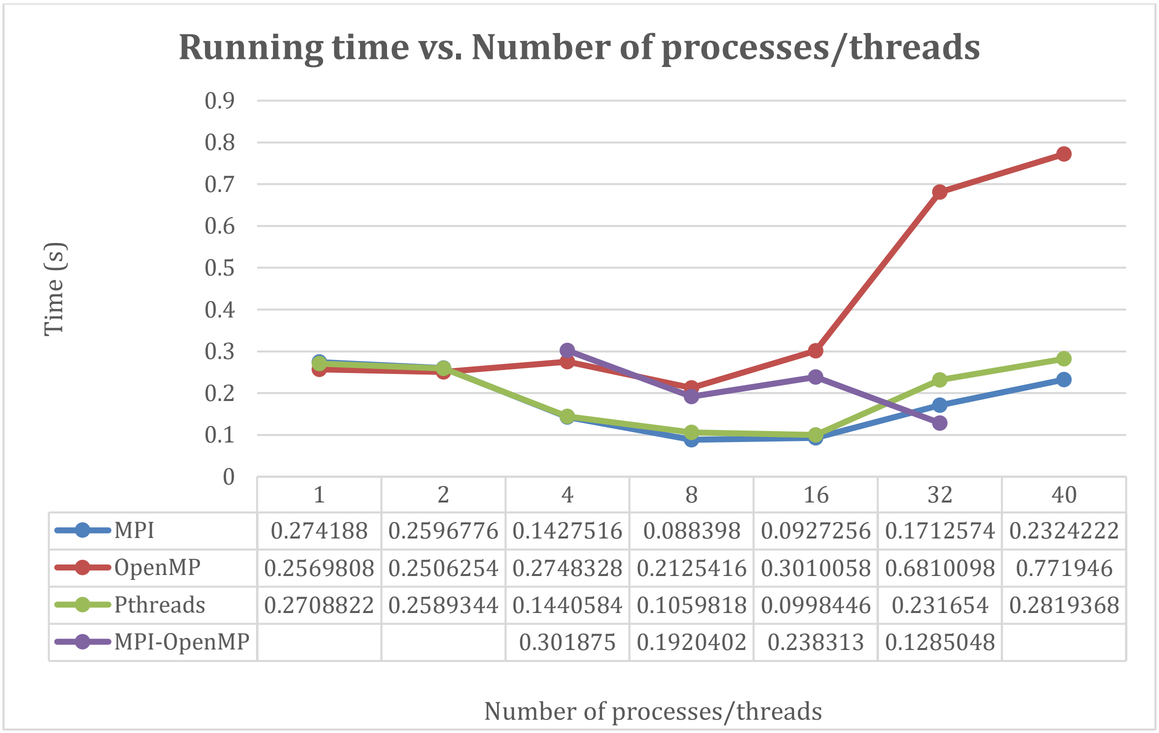
```
❯ ./sequential 200 10
Iteration 0, elapsed time: 0.002
Iteration 1, elapsed time: 0.002
Iteration 2, elapsed time: 0.002
Iteration 3, elapsed time: 0.002
Iteration 4, elapsed time: 0.003
Iteration 5, elapsed time: 0.003
Iteration 6, elapsed time: 0.002
Iteration 7, elapsed time: 0.002
Iteration 8, elapsed time: 0.002
Iteration 9, elapsed time: 0.002
Total time: 0.064781
Student ID: 119020059
Name: Xinyu Xie
Assignment 2: N Body Simulation Sequential Implementation
Number of Bodies: 200
Number of Cores: 1
```

Analysis and Results

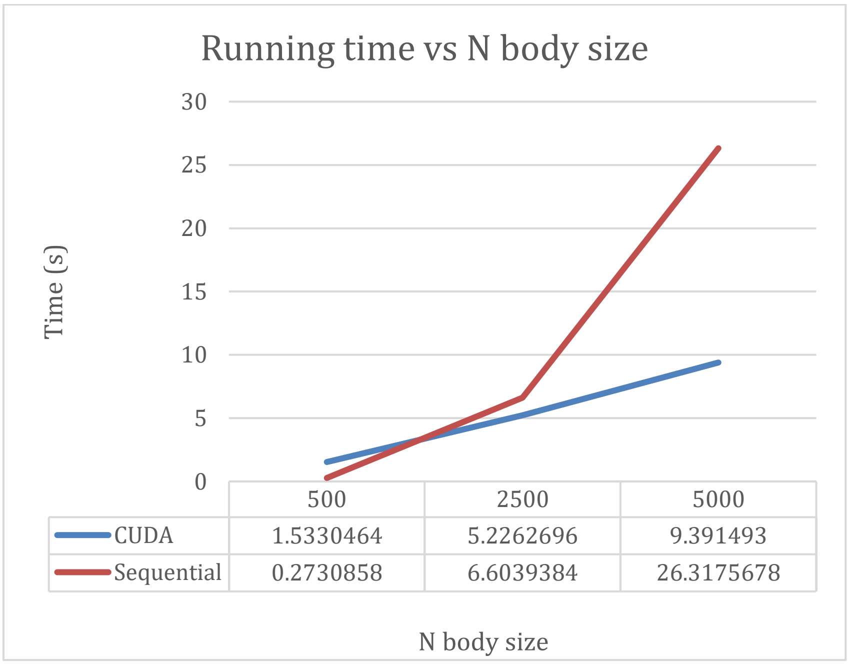
Running time vs. Number of processes/threads

Fix number of bodies = 500, number of iterations = 100. We compare parallel version of MPI, Pthreads, OpenMP with number of processes/threads = 1, 2, 4, 8, 16, 32, 40. For MPI + OpenMP version, I only adjust number of processes with fixed OpenMP threads number = 4, thus MPI + OpenMP has number of processes/threads

= 4, 8, 16, 32. For CUDA version, since the number of threads is the smallest multiple of 512 that is larger than N bodies, I analyze the relationship between running time and number of threads for CUDA version alone. The comparison is shown as follows:



	1	2	4	8	16	32	40
MPI	0.274188	0.2596776	0.1427516	0.088398	0.0927256	0.1712574	0.2324222
OpenMP	0.2569808	0.2506254	0.2748328	0.2125416	0.3010058	0.6810098	0.771946
Pthreads	0.2708822	0.2589344	0.1440584	0.1059818	0.0998446	0.231654	0.2819368
MPI-OpenMP			0.301875	0.1920402	0.238313	0.1285048	



	500	2500	5000
CUDA	1.5330464	5.2262696	9.391493
Sequential	0.2730858	6.6039384	26.3175678

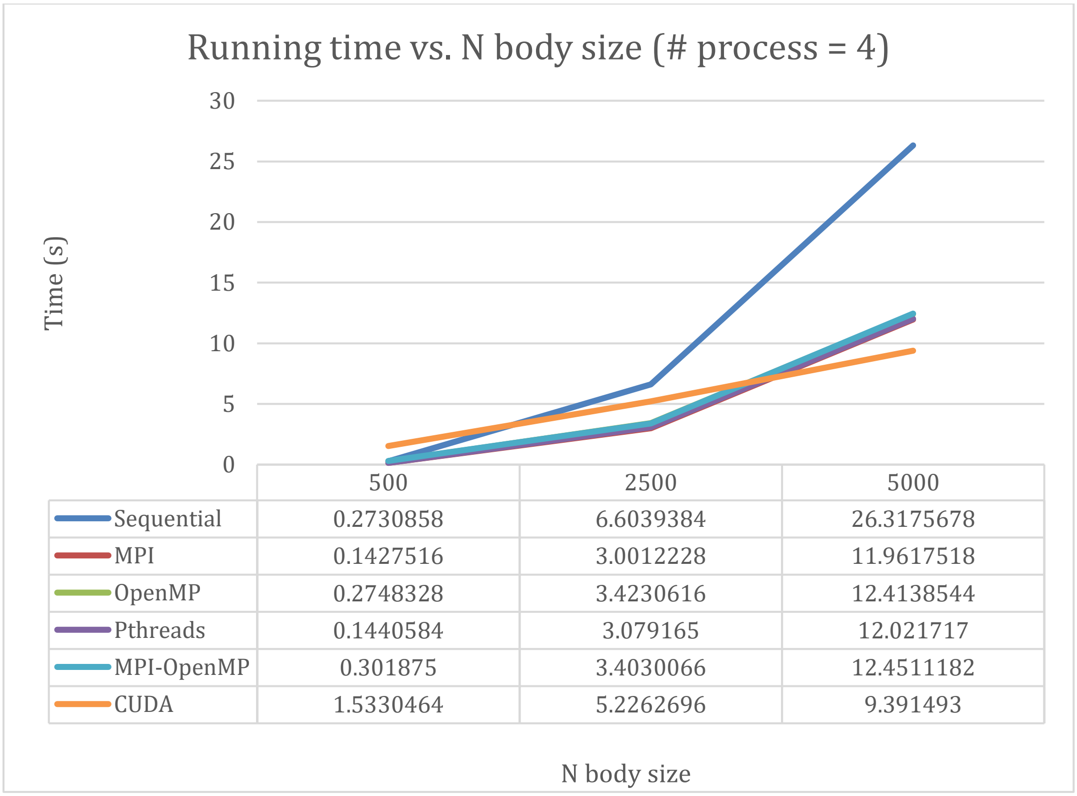
The above images show all parallel versions implementation of the relationship between running time and number of processes/threads. From the figure above, for MPI, OpenMP, Pthreads versions, I found that when the number of processes/threads increases, the running time firstly decreases but then increases.

For MPI-OpenMP version, the running time generally decreases with slight fluctuations as number of processes/threads increases.

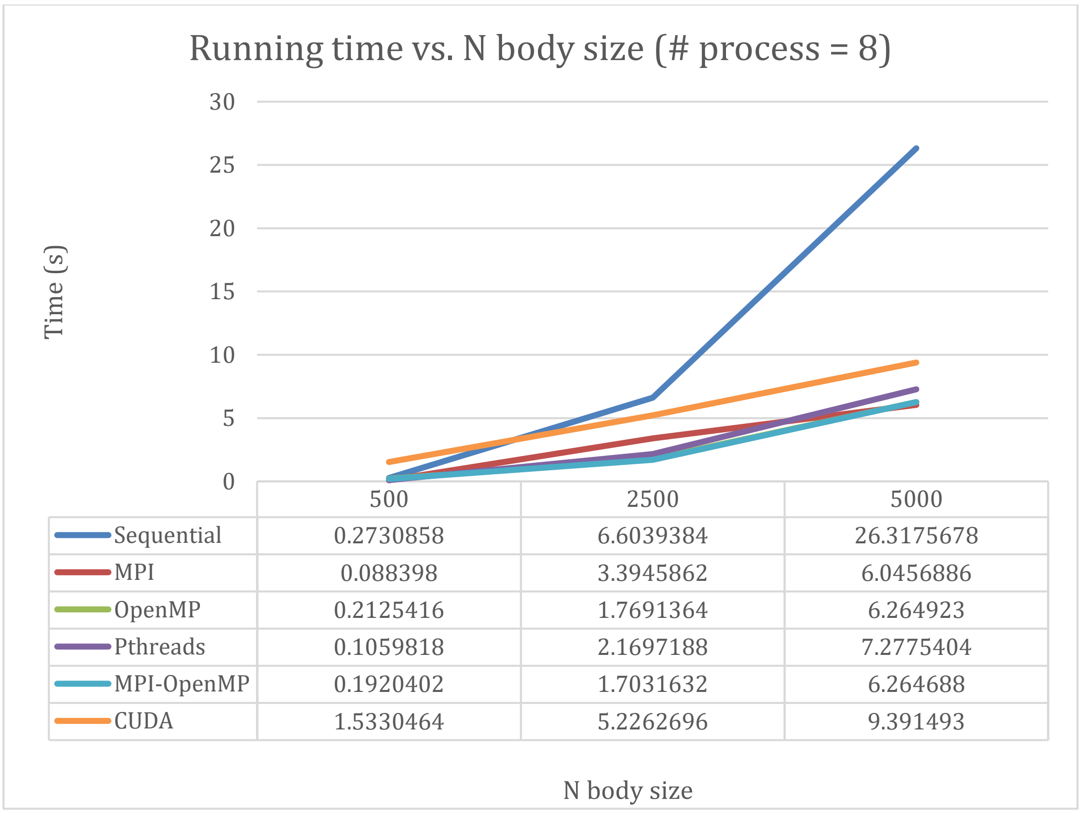
For MPI, OpenMP, Pthreads versions, initially, when the number of processes/threads increases, the computation workload in each process/thread is less than before, thus firstly the runnning time decreases, which illustrates that parallel versions of the program indeed reduce running time. However, when the number of processes/threads is very large, the costs for process communication and synchronization increases as the number of processes/threads increases. Since the costs for communication and synchronization dominate computation time when number of process/threads is very large, the running time increases as the number of processes/threads increases. For CUDA version, number of threads is just N body size. Since each thread is only responsible for updating one body in each iteration, the main cost lies in memory copy, which heavily dominate local computation time in each thread. Thus, as number of threads increases, the running time of CUDA version is not very small compared to its thread number. For MPI + OpenMP version, when number of processes/threads increases, the running time generally decreases since with smaller workload, each process/thread needs less local computation time.

Sequential vs. MPI vs. Pthread vs. CUDA vs. OpenMP

Fix number of iterations = 100, fix number of processes/threads = 4 or 8 for MPI, Pthreads, OpenMP, MPI + OpenMP versions. We compare sequential, MPI, Pthreads, OpenMP, MPI + OpenMP, CUDA versions with N body size = 500, 2500, 5000. The comparison is shown as follows:



	500	2500	5000
Sequential	0.2730858	6.6039384	26.3175678
MPI	0.1427516	3.0012228	11.9617518
OpenMP	0.2748328	3.4230616	12.4138544
Pthreads	0.1440584	3.079165	12.021717
MPI-OpenMP	0.301875	3.4030066	12.4511182
CUDA	1.5330464	5.2262696	9.391493



	500	2500	5000
Sequential	0.2730858	6.6039384	26.3175678
MPI	0.088398	3.3945862	6.0456886
OpenMP	0.2125416	1.7691364	6.264923
Pthreads	0.1059818	2.1697188	7.2775404
MPI-OpenMP	0.1920402	1.7031632	6.264688
CUDA	1.5330464	5.2262696	9.391493

The above images show the sequential, MPI, Pthreads, OpenMP, MPI + OpenMP, CUDA versions of the relationship between running time and N body size with fixed number of processes/threads. From the figure above, I found that keeping number of processes/threads constant, when the N body size is small, the running time is less. Since when N body size is large, total number of bodies needed to be computed is larger, thus longer computation time is needed. The running time is roughly linear with N body size from the graph.

Moreover, I found that parallel versions have much less running time compared with sequential version. This illustrates that parallel computing could indeed reduce much computation time. The running time of MPI, Pthread, OpenMP, MPI + OpenMP versions is roughly the same when number of processes equals to number of threads. The running time is smaller for CUDA versions when the N body size is large enough. Since when the N body size is large enough, the cuda memory copy time is small, with much more number of threads than other parallel versions, the running time of CUDA should be much smaller. The slight fluntuations of running time maybe due to the costs of process communication and thread synchronization of different parallel versions of implementation.

Conclusion

In this project, the sequential version, parallel version using MPI, parallel version using Pthread, parallel version using OpenMP, parallel version using CUDA, and parallel version using MPI + OpenMP, of N-body simulation are implemented and their proformances are also analyzed.

We can conclude from this project that when the number of processes/threads increases within reasonable range, the running time decreases. However, when the number of processes/threads is too large, the running time may increases due to large costs of process communication and thread synchrnozation. Moreover, keeping number of processes/threads constant, when the N body size is small, the running time is less. The running time is roughly a linear function of N body size.

Through this project, I learned much knowledge regarding the parallel programming and also how to use the MPI, Pthread, OpenMP, CUDA, and MPI + OpenMP methods to implement parallel programs. I also enjoy a lot when conducting experiments for different number of bodies, different versions of implementation, and exploring the relationship between the running time and the number of processes/threads.