Assignment Report

CSC 4005 N-body Simulation

Wei Wu (吴畏)

118010335

November 15, 2021

The School of Data Science

香 港 中 文 大 學（深 圳）
The Chinese University of Hong Kong, Shenzhen

I. Introduction

This assignment is implementing a sequential program, a P-thread program, an CUDA program, a openMP program and an MPI program to simulate an astronomical N-body system, but in two-dimensions.

The bodies are initially at rest. Their initial positions and masses are to be selected randomly (using a random number generator). The gravity between N-body should be described by the following equation:

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

Also we should consider the collision and bouncing, otherwise, all the points will be collasped into a singular point. Display the movement of the bodies using xlib or other GUI systems on your computers.

II. Design Approaches

1. Architecture

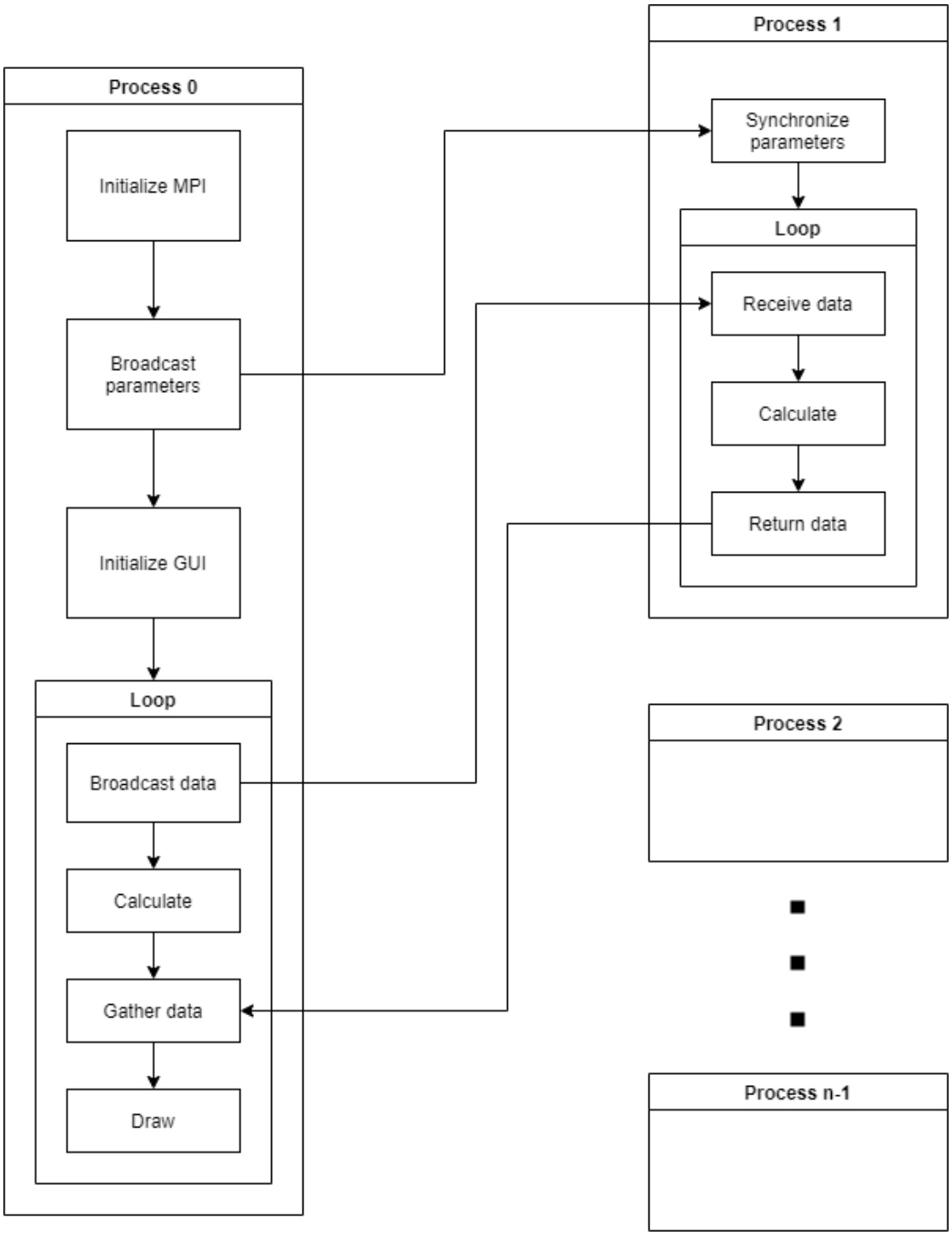

Figure 1. Architecture of the MPI version of the program

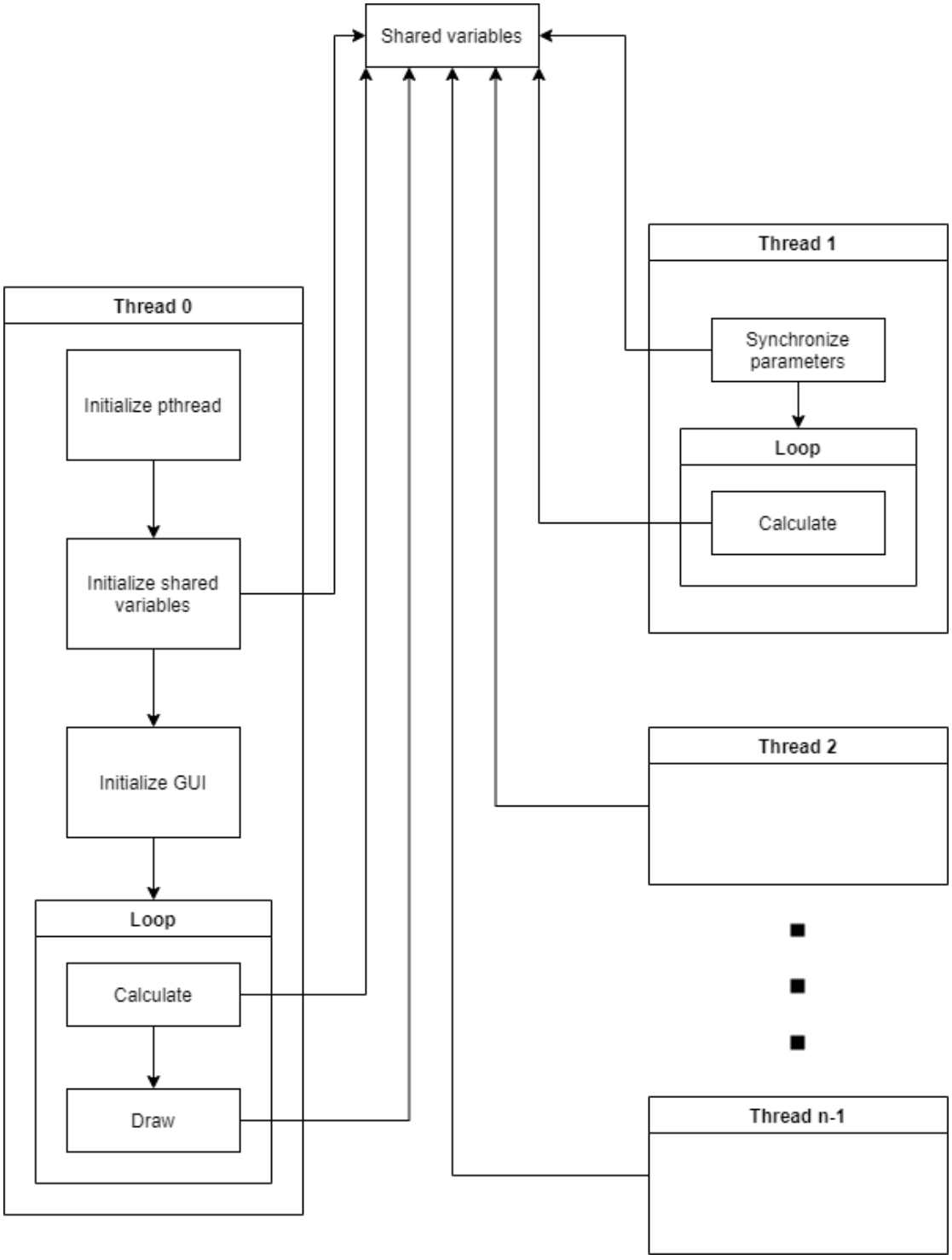

Figure 2. Architecture of the pthread version of the program

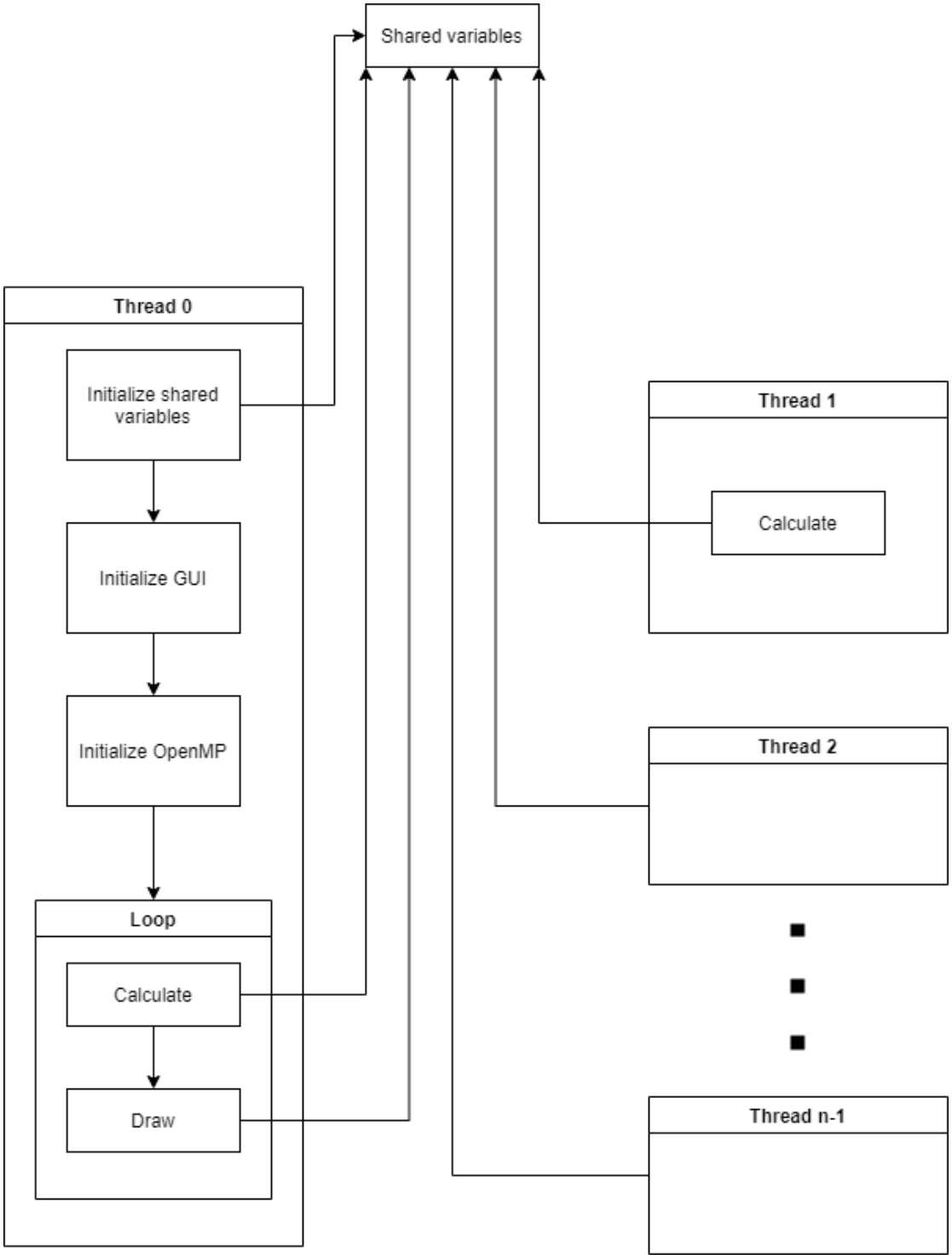

Figure 3. Architecture of the OpenMP version of the program

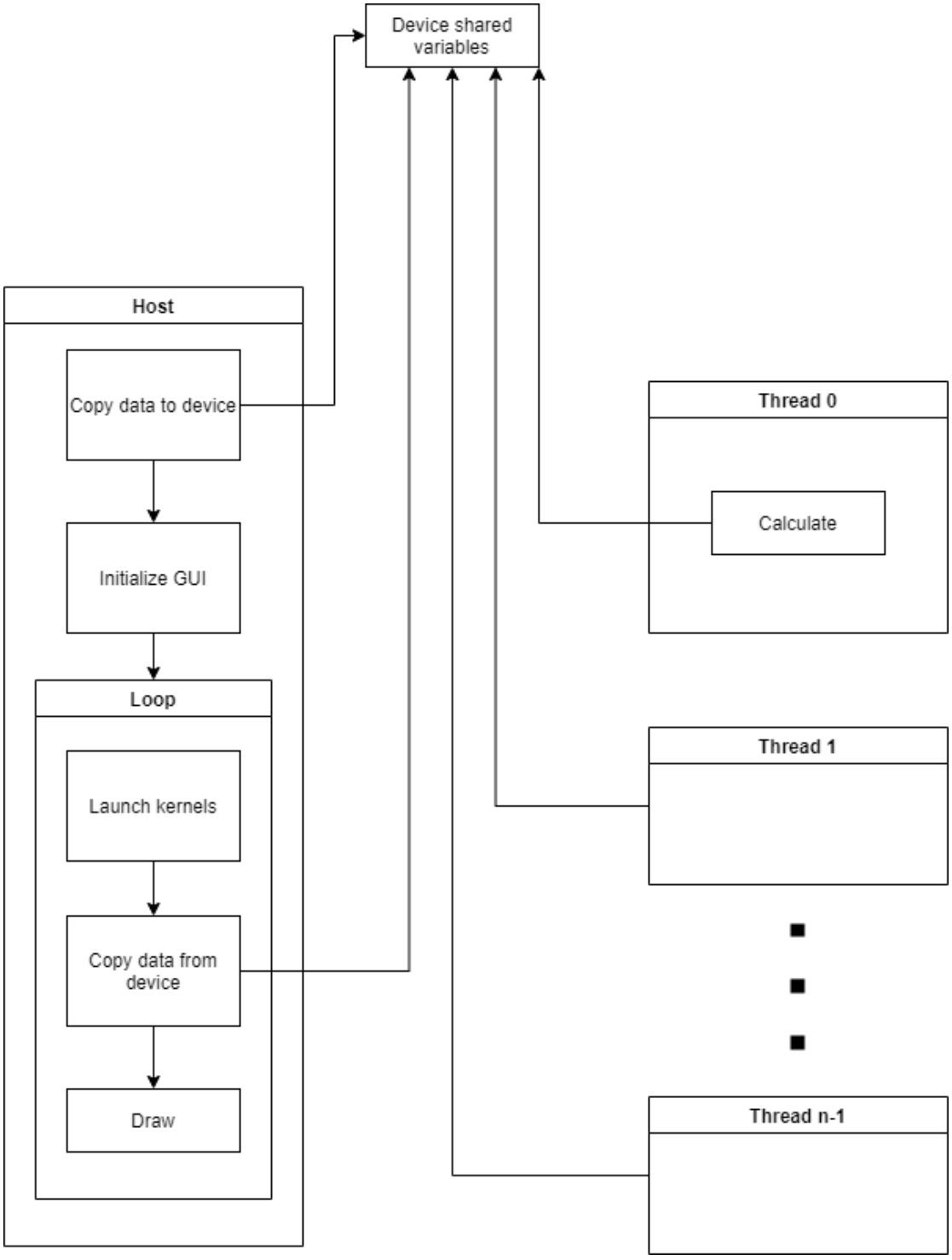

Figure 4. Architecture of the CUDA version of the program

III. Build And Run

1. MPI

To build the program, use the following commands on the server:

cd /path/to/project

mkdir build && cd build

cmake .. -DCMAKE_BUILD_TYPE=Release

make

cp ~/.Xauthority /pvfsmnt/$(whoami)

export XAUTHORITY=/pvfsmnt/$(whoami)/.Xauthority

(1) To run the program, use the following commands on the server:

mpirun csc4005_imgui (gravity) (bodies) (elapse) (max_iteration)

Examples:

a. mpirun csc4005_imgui

b. mpirun csc4005_imgui 100 100 0.1 100

2. pthread

To build the program, use the following commands on the server:

cd /path/to/project

mkdir build && cd build

cmake .. -DCMAKE_BUILD_TYPE=Release

make

cp ~/.Xauthority /pvfsmnt/$(whoami)

export XAUTHORITY=/pvfsmnt/$(whoami)/.Xauthority

(2) To run the program, use the following commands on the server:

mpirun csc4005_imgui num_threads (gravity) (bodies) (elapse) (max_iteration)

Examples:

a. srun -n1 csc4005_imgui 4

b. srun -n1 csc4005_imgui 8 100 100 0.1 100

3. OpenMP

To build the program, use the following commands on the server:

cd /path/to/project

mkdir build && cd build

cmake .. -DCMAKE_BUILD_TYPE=Release

make

cp ~/.Xauthority /pvfsmnt/$(whoami)

export XAUTHORITY=/pvfsmnt/$(whoami)/.Xauthority

(3) To run the program, use the following commands on the server:

mpirun csc4005_imgui num_threads (gravity) (bodies) (elapse) (max_iteration)

Examples:

a. srun -n1 csc4005_imgui 4

b. srun -n1 csc4005_imgui 8 100 100 0.1 100

4. CUDA

To build the program, use the following commands on the server:

cd /path/to/project

mkdir build && cd build

source scl_source enable devtoolset-10

CC=gcc CXX=g++ cmake ..

make -j12

cp ~/.Xauthority /pvfsmnt/$(whoami)

export XAUTHORITY=/pvfsmnt/$(whoami)/.Xauthority

(4) To run the program, use the following commands on the server:

mpirun csc4005_imgui num_threads (gravity) (bodies) (elapse) (max_iteration)

Examples:

c. srun -n1 csc4005_imgui 4

d. srun -n1 csc4005_imgui 8 100 100 0.1 100

IV. Performance Analysis

1. Test Results

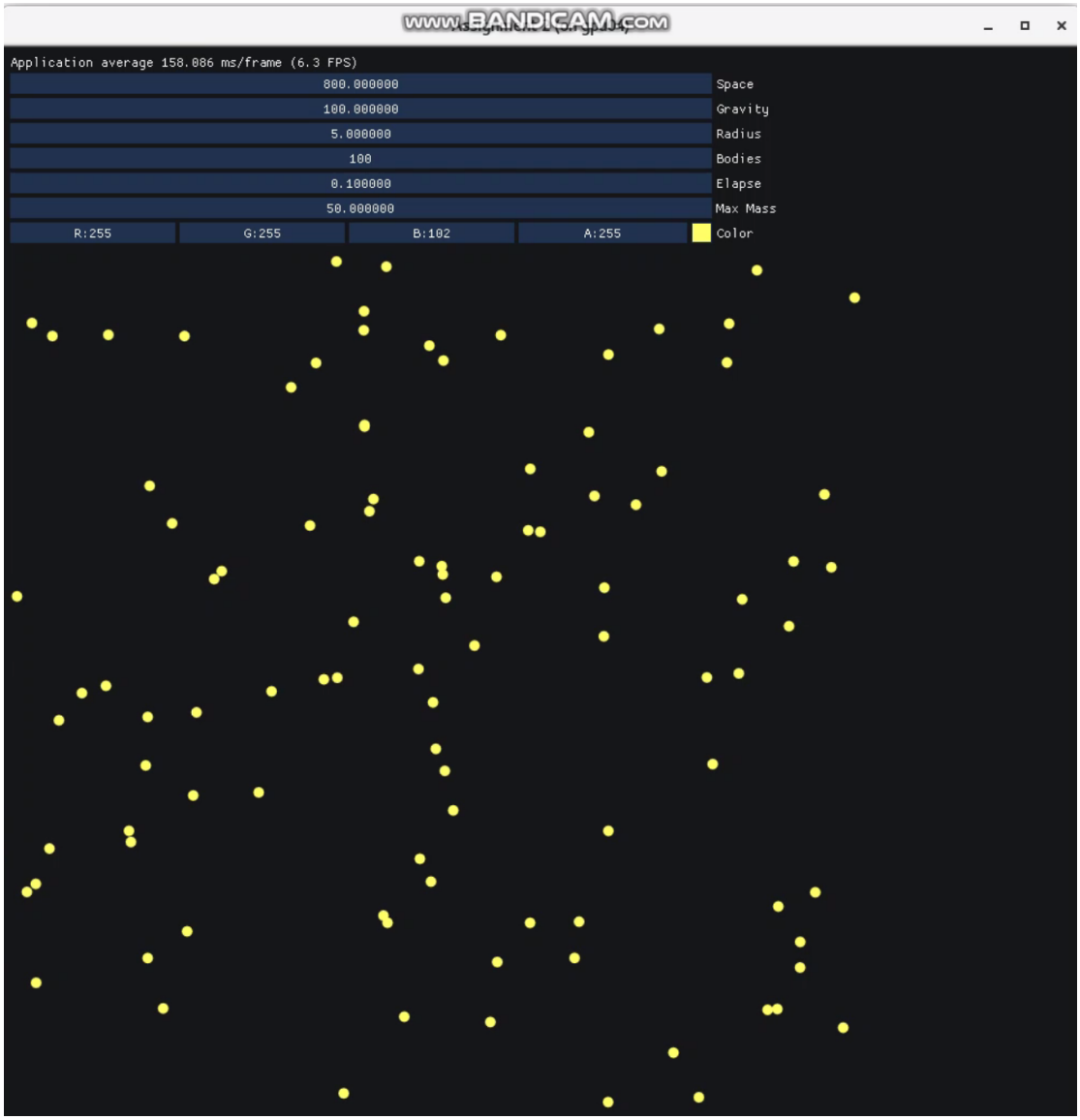

Figure 6. Test result with bodies = 100

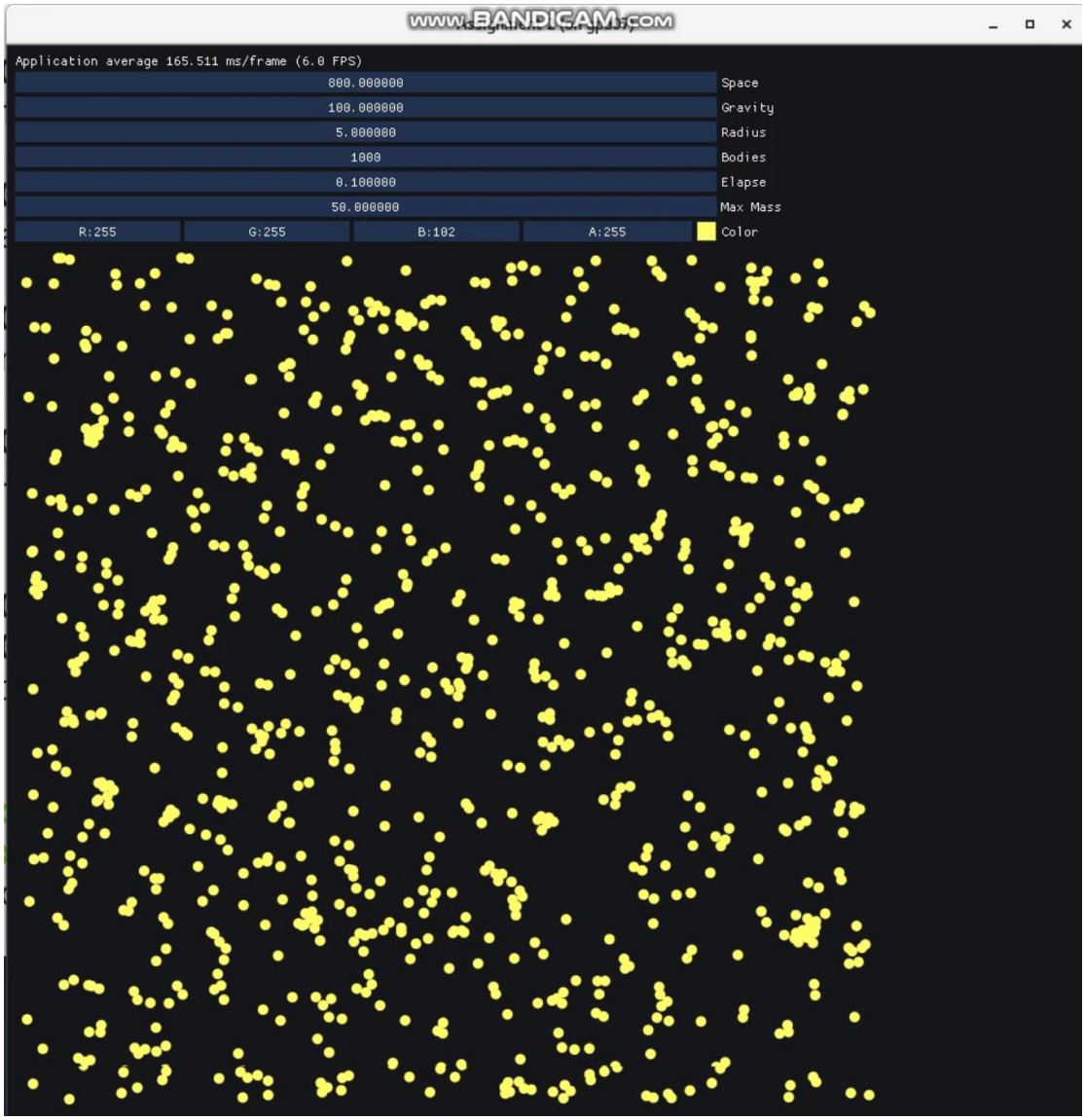

Figure 7. Test result with bodies = 1000

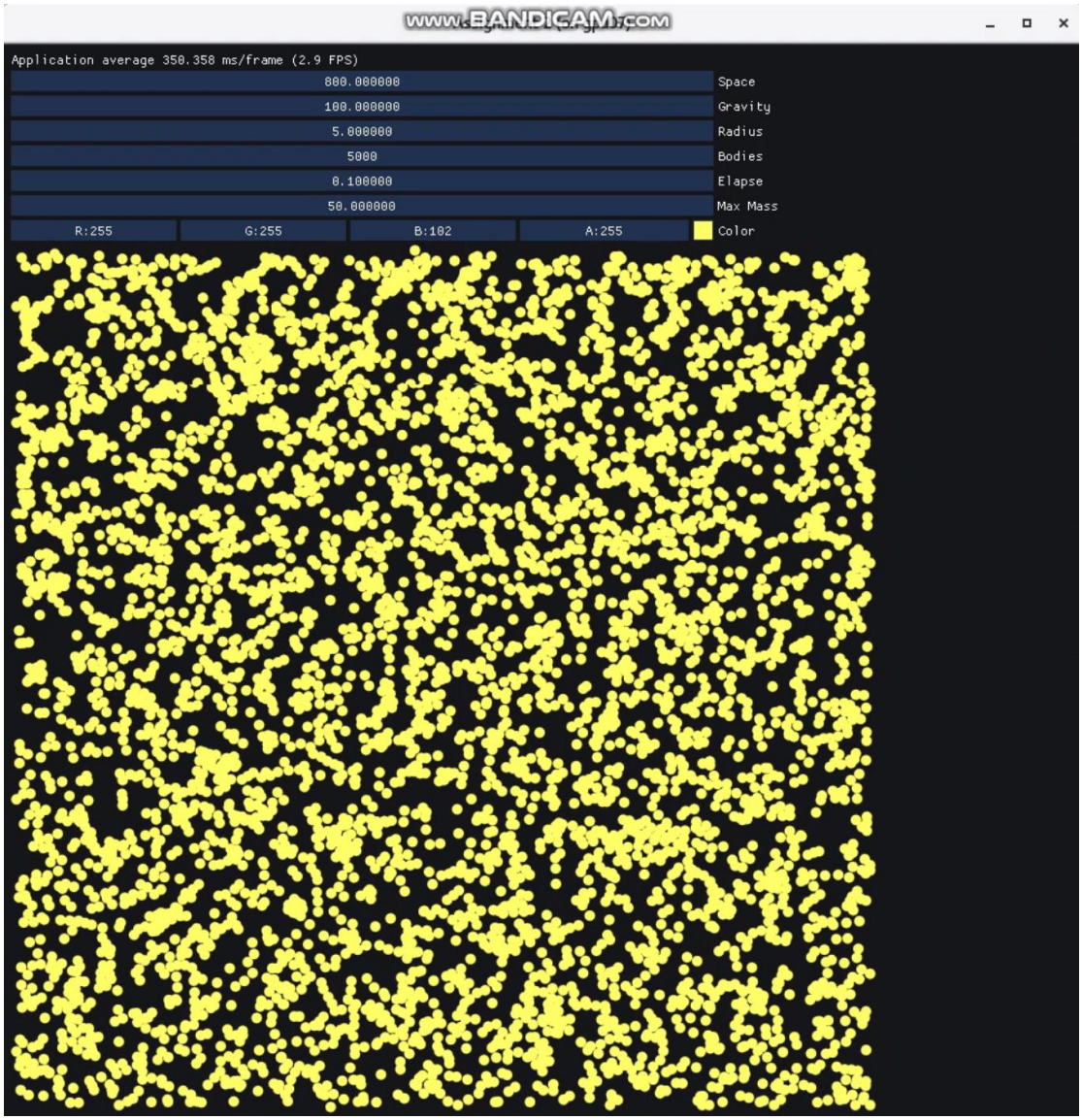

Figure 8. Test result with bodies = 5000

bodies	Number of processes					
		1	4	16	32	64
	100	622	1202	1517	1000	136.3
	1000	64	252	878	837	569
	5000	13	51	201	198	226

Table 1. Calculation speed (bodies / ms) of the MPI version of the program with respect to different numbers of bodies and numbers of processes

	Number of processes				
bodies	1	4	16	32	64
100	923	1410	573	334	131
1000	100	394	1299	1396	565
5000	20	81	321	595	971

Table 2. Calculation speed (bodies / ms) of the pthread version of the program with respect to different numbers of bodies and numbers of processes

	Number of processes				
bodies	1	4	16	32	64
100	874	2075	6312	2797	2414
1000	93	361	1371	2408	4822
5000	19	75	297	564	1149

Table 3. Calculation speed (bodies / ms) of the OpenMP version of the program with respect to different numbers of bodies and numbers of processes

	Number of processes				
bodies	1	4	16	32	64
100	6.87	27	90.75	149.66	214.34
1000	0.98	3.84	13.32	21.45	32.12
5000	0.19	0.6	2.95	5.88	11.22

Table 4. Calculation speed (bodies / ms) of the CUDA version of the program with respect to different numbers of bodies and numbers of processes

Note:

a. All tests were conducted on 10.26.1.30.

b. Each test was repeated for three times and averaged.

c. All input data were generated randomly.

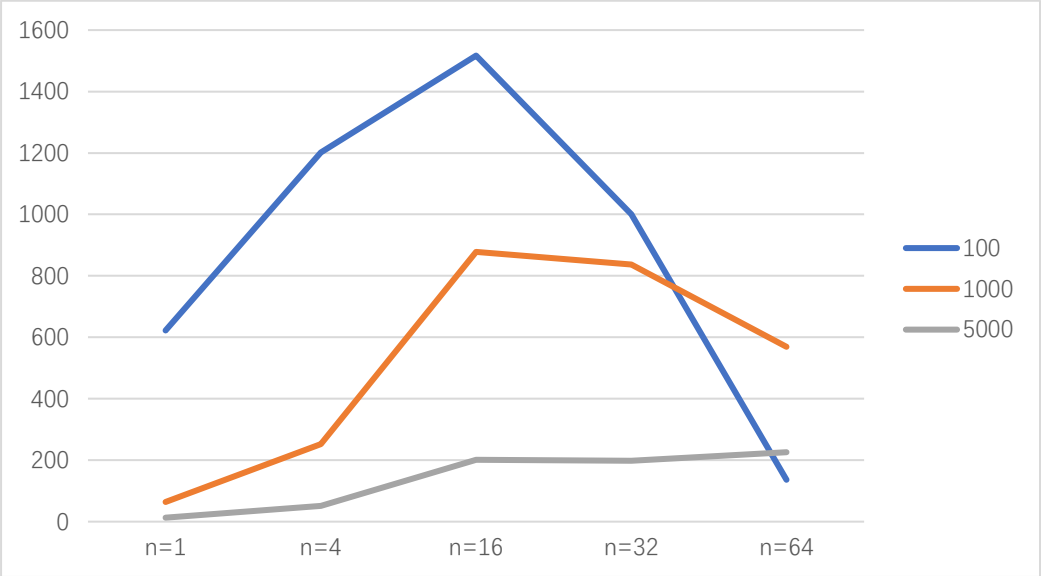

Figure 9. Calculation speed (bodies / ms) of the MPI version of the program with respect to different number of processes

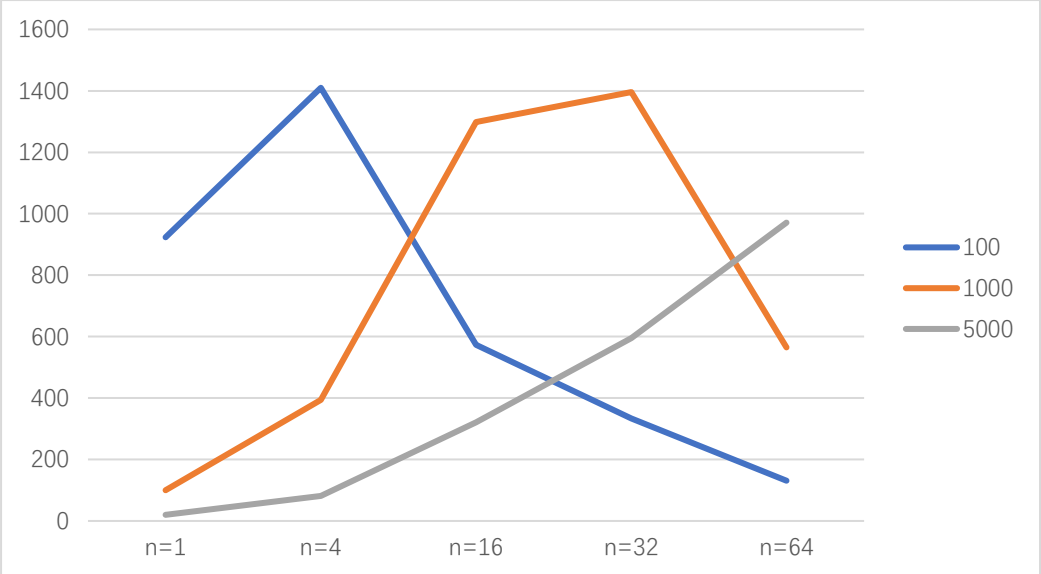

Figure 10. Calculation speed (bodies / ms) of the pthread version of the program with respect to different number of processes

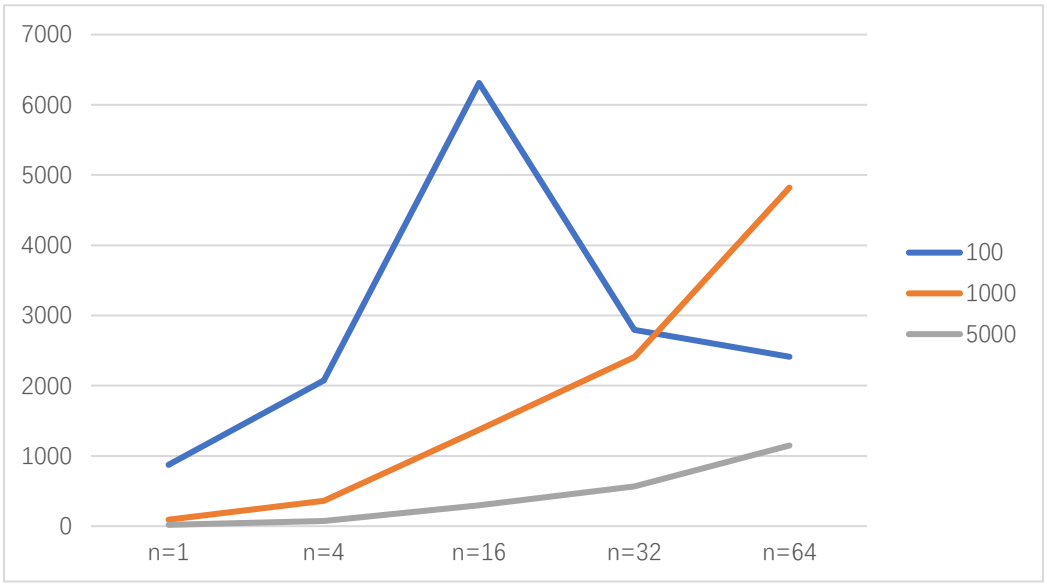

Figure 11. Calculation speed (bodies / ms) of the OpenMP version of the program with respect to different number of processes

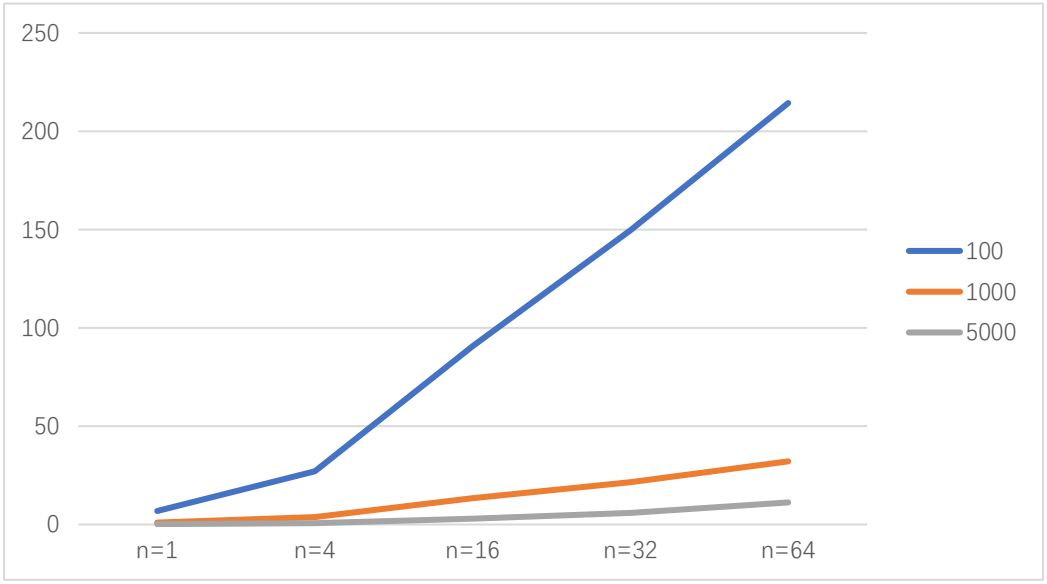

Figure 12. Calculation speed (bodies / ms) of the CUDA version of the program with respect to different number of processes

2. Analysis

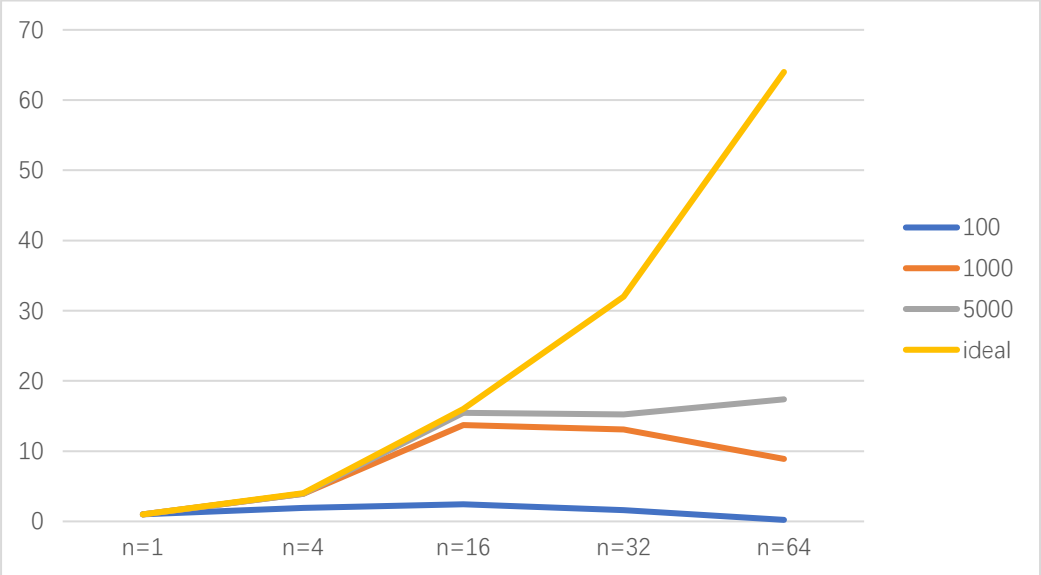

Figure 14. Speedup factor of the MPI version of the program with respect to different number of processes

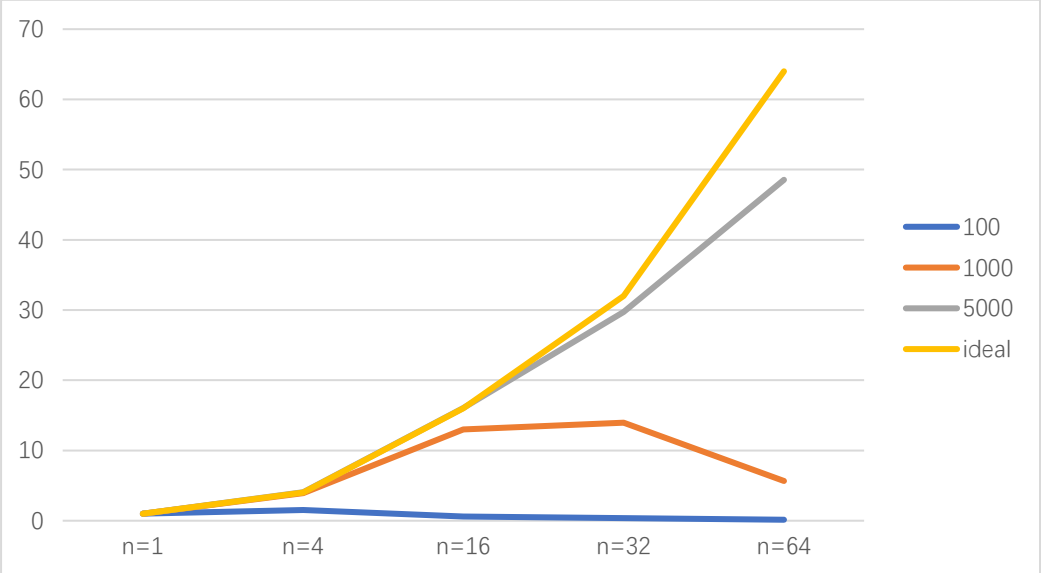

Figure 15. Speedup factor of the pthread version of the program with respect to different number of processes

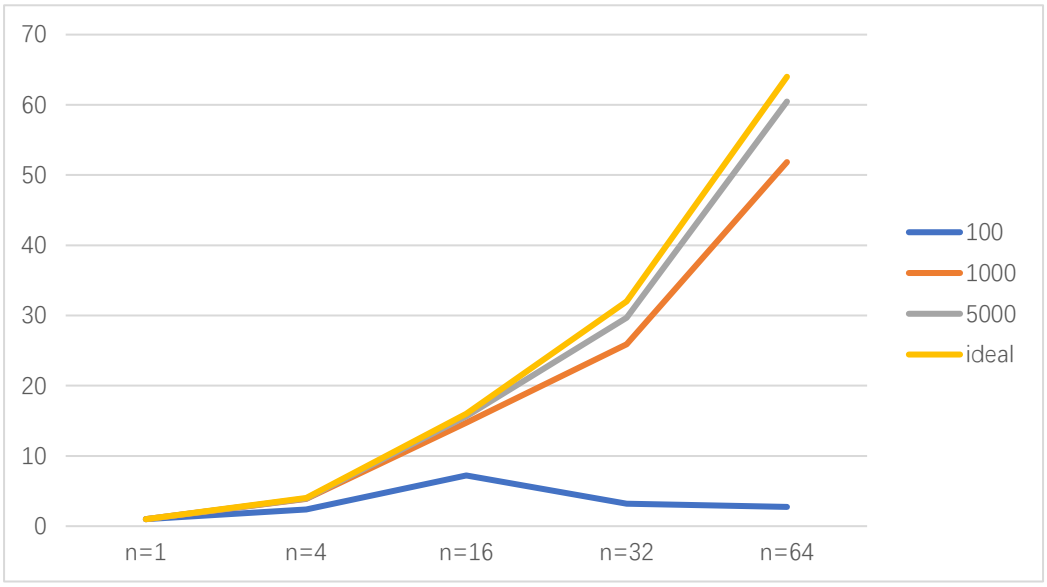

Figure 16. Speedup factor of the OpenMP version of the program with respect to different number of processes

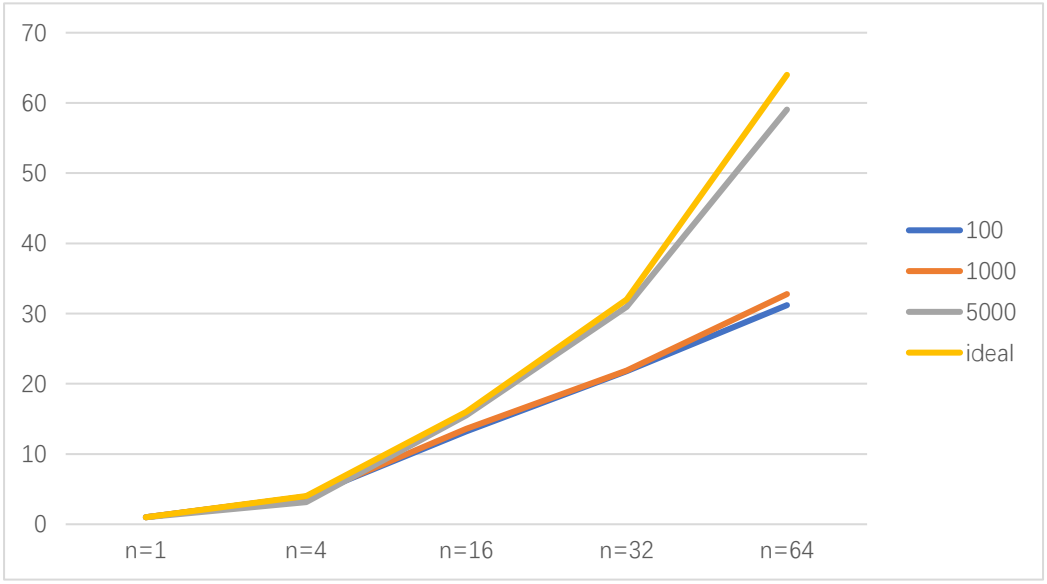

Figure 17. Speedup factor of the CUDA version of the program with respect to different number of processes

As shown in Figure 14-18, when the number of bodies is small (100) and the number of processes/threads is also small, the calculation speed does not increase much as the number of processes/threads increases.

When the number of processes/threads becomes larger, the calculation speed even drops due to the multiprocessing overhead, such as inter-process communication, shared memory access, mutual exclusive locks, etc.

Generally speaking, the speedup is better for larger number of bodies. However, in the MPI version of the program, the speedup is less than 20 when the number of processes is 64. The reason might be that the inter-process communication overhead (synchronizing the x, y, vx, vy data in each iteration) gets larger as the number of bodies increases. The other versions of the program use the shared memory mechanism, which avoids the data transfer overhead.

In terms of performance, the OpenMP version generally provides the best calculation speed. Notice that the CUDA version seems to be very slow. There are a few possible reasons. First, the timing function is executed on the CPU. Therefore, the measured calculation time may be much larger than the actual calculation time on the GPU. Second, a single CUDA core may be much slower than a CPU core.

Nevertheless, this does not necessarily mean that the OpenMP framework is superior than the others. There are many factors that may affect the performance. For example, my design and implementation of each version of the program. The choice of the parallel programming framework should depend on the specific usage (CPU-bound or IO-

bound), algorithm, and hardware.

In all versions, it is obvious that as the number of processes increases, the speedup factor increases slower and even decreases for small input sizes, while the efficiency decreases.

V. Conclusion

In summary, this assignment explores the N-body Simulation algorithm and implementations by different parallel computing frameworks, the concepts and implementation of parallel computing, the MPI, pthread, OpenMP, and CUDA framework, and the speedup and overhead of multiprocessing/multithreading. Due to the inter-process communication and/or the shared memory access overhead, when the number of processes becomes larger, the efficiency of multiprocessing/multithreading gets lower. For small input sizes, this overhead may even outweigh the speedup from parallelism when the number of processes/threads is large. However, for large input sizes, the impact of the multiprocessing/multithreading overhead is relatively smaller since the portion of computation is larger. Therefore, parallel computing is best suitable for large input sizes.

There exist some limitations in this assignment. First, since the maximum time of a single session of the cloud server is 10 minutes, it is not feasible to test the program for a long time to see whether the calculation speed is stable over time. By observation, in the pthread, OpenMP, and CUDA implementation, when there is a collision between two bodies, one or two mutual exclusive locks must be used. This implies that when there are many collisions in one iteration, processes must wait for each other to avoid data race. Under this circumstance, the overall

calculation speed may degrade by a lot.

Second, since the maximum processor cores of a single session of the cloud server is 64, it is not feasible to test larger number of cores. Nevertheless, from the existing experiment results, we can infer that when the input size is large enough, the efficiency of multiprocessing will be close to 1. Also, we can infer that the multiprocessing overhead will become larger and larger as the number of processors increases, and finally slows down the execution.

VI. Source Code

1. MPI Version

```
#include <graphic/graphic.hpp>
#include <imgui_impl_sdl.h>
#include <cstring>
#include <nbody/body.hpp>
#include <mpi.h>
#include <chrono>
#include <iostream>
#include <iomanip>
#include <cstring>
#include <cstdint>
#include <cstddef>

template <typename ...Args>
void UNUSED(Args&&... args [[maybe_unused]]) {}

int main(int argc, char **argv) {

    int size;
    int rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int num_iteration = 0;
    int max_iteration = 100;
    size_t duration = 0;
    size_t bodies_count = 0;

    static float gravity = 100;
    static float space = 800;
    static float radius = 5;
    static int bodies = 20;
    static float elapse = 0.001;
    static ImVec4 color = ImVec4(1.0f, 1.0f, 0.4f, 1.0f);
    static float max_mass = 50;

    if (argc >= 2) {
        gravity = atof(argv[1]);
    }
    if (argc >= 3) {
```

```
        bodies = atof(argv[2]);
    }
    if (argc >= 4) {
        elapse = atof(argv[3]);
    }
    if (argc >= 5) {
        max_iteration = atoi(argv[4]);
    }

    static float current_space = space;
    static float current_max_mass = max_mass;
    static int current_bodies = bodies;

    double* x;
    double* y;
    double* vx;
    double* vy;

    int chunk = (bodies + size - 1) / size;

    // The process-local bodypool
    BodyPool* local_pool = new BodyPool(static_cast<size_t>(bodies),
static_cast<size_t>(chunk * size), space, max_mass);

    // Max index for each process
    int i_max = 0;
    if (chunk * (rank + 1) > (int) local_pool->size()) {
        i_max = (int) local_pool->size();
    }
    else {
        i_max = chunk * (rank + 1);
    }
    if (0 == rank) {
        // Broadcast the mass
        MPI_Bcast(local_pool->m.data(), (int) local_pool->size(),
MPI_DOUBLE, 0, MPI_COMM_WORLD);
        // Buffer for MPI gather
        x = new double[chunk * size];
        y = new double[chunk * size];
        vx = new double[chunk * size];
        vy = new double[chunk * size];
        graphic::GraphicContext context{"Assignment 2"};
        context.run([&](graphic::GraphicContext *context
[[maybe_unused]], SDL_Window *) {
```

```
            auto io = ImGui::GetIO();
            ImGui::SetNextWindowPos(ImVec2(0.0f, 0.0f));
            ImGui::SetNextWindowSize(io.DisplaySize);
            ImGui::Begin("Assignment 2", nullptr,
                         ImGuiWindowFlags_NoMove
                         | ImGuiWindowFlags_NoCollapse
                         | ImGuiWindowFlags_NoTitleBar
                         | ImGuiWindowFlags_NoResize);
            ImDrawList *draw_list = ImGui::GetWindowDrawList();
            ImGui::Text("Application average %.3f ms/frame (%.1f FPS)",
1000.0f / ImGui::GetIO().Framerate,
                        ImGui::GetIO().Framerate);
            ImGui::DragFloat("Space", &current_space, 10, 200, 1600,
"%f");
            ImGui::DragFloat("Gravity", &gravity, 0.5, 0, 1000, "%f");
            ImGui::DragFloat("Radius", &radius, 0.5, 2, 20, "%f");
            ImGui::DragInt("Bodies", &current_bodies, 1, 2, 100, "%d");
            ImGui::DragFloat("Elapse", &elapse, 0.001, 0.001, 10, "%f");
            ImGui::DragFloat("Max Mass", &current_max_mass, 0.5, 5, 100,
"%f");
            ImGui::ColorEdit4("Color", &color.x);

            // Parameter adjustment in GUI is disabled
            auto begin = std::chrono::high_resolution_clock::now();

            if (num_iteration < max_iteration) {
                // Broadcast the location and velocity
                MPI_Bcast(local_pool->x.data(), (int)
local_pool->size(), MPI_DOUBLE, 0, MPI_COMM_WORLD);
                MPI_Bcast(local_pool->y.data(), (int)
local_pool->size(), MPI_DOUBLE, 0, MPI_COMM_WORLD);
                MPI_Bcast(local_pool->vx.data(), (int)
local_pool->size(), MPI_DOUBLE, 0, MPI_COMM_WORLD);
                MPI_Bcast(local_pool->vy.data(), (int)
local_pool->size(), MPI_DOUBLE, 0, MPI_COMM_WORLD);
                for (size_t i = rank * chunk; i < i_max; ++i) {
                    local_pool->ax[i] = 0;
                    local_pool->ay[i] = 0;
                    for (size_t j = 0; j < local_pool->size(); ++j) {
                        if (j == i) continue;
                        local_pool->check_and_update_mpi(local_pool->get_
body(i), local_pool->get_body(j), radius, gravity);
                    }
                }
```

```
                for (size_t i = rank * chunk; i < i_max; ++i) {
                    local_pool->get_body(i).update_for_tick(elapse,
space, radius);
                }
                // Gather the location and velocity
                MPI_Gather(&local_pool->x[chunk * rank], chunk,
MPI_DOUBLE, x, chunk, MPI_DOUBLE, 0, MPI_COMM_WORLD);
                MPI_Gather(&local_pool->y[chunk * rank], chunk,
MPI_DOUBLE, y, chunk, MPI_DOUBLE, 0, MPI_COMM_WORLD);
                MPI_Gather(&local_pool->vx[chunk * rank], chunk,
MPI_DOUBLE, vx, chunk, MPI_DOUBLE, 0, MPI_COMM_WORLD);
                MPI_Gather(&local_pool->vy[chunk * rank], chunk,
MPI_DOUBLE, vy, chunk, MPI_DOUBLE, 0, MPI_COMM_WORLD);
                for (int i = 0; i < local_pool->size(); i++) {
                    local_pool->x[i] = x[i];
                    local_pool->y[i] = y[i];
                    local_pool->vx[i] = vx[i];
                    local_pool->vy[i] = vy[i];
                }
                num_iteration ++;
                bodies_count += bodies;
            }
            auto end = std::chrono::high_resolution_clock::now();
            duration += duration_cast<std::chrono::nanoseconds>(end -
begin).count();
            // Only print the result once
            static bool isResultPrinted = false;
            if (!isResultPrinted && num_iteration >= max_iteration) {
                std::cout << bodies_count << " bodies in last " <<
duration << " nanoseconds\n";
                auto speed = static_cast<double>(bodies_count) /
static_cast<double>(duration) * 1e6;
                std::cout << "speed: " << std::setprecision(12) << speed
<< " bodies per millisecond" << std::endl;
                isResultPrinted = true;
            }
            // Draw the balls
            {
                const ImVec2 p = ImGui::GetCursorScreenPos();
                for (size_t i = 0; i < local_pool->size(); ++i) {
                    auto body = local_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});
                }
            }

            ImGui::End();
        });
    }

    else {
        // Receive the mass
        MPI_Bcast(local_pool->m.data(), (int) local_pool->size(),
MPI_DOUBLE, 0, MPI_COMM_WORLD);
        for (int it = 0; it < max_iteration; it++) {
            // Synchronize the location and velocity
            MPI_Bcast(local_pool->x.data(), (int) local_pool->size(),
MPI_DOUBLE, 0, MPI_COMM_WORLD);
            MPI_Bcast(local_pool->y.data(), (int) local_pool->size(),
MPI_DOUBLE, 0, MPI_COMM_WORLD);
            MPI_Bcast(local_pool->vx.data(), (int) local_pool->size(),
MPI_DOUBLE, 0, MPI_COMM_WORLD);
            MPI_Bcast(local_pool->vy.data(), (int) local_pool->size(),
MPI_DOUBLE, 0, MPI_COMM_WORLD);
            for (size_t i = rank * chunk; i < i_max; ++i) {
                local_pool->ax[i] = 0;
                local_pool->ay[i] = 0;
                for (size_t j = 0; j < local_pool->size(); ++j) {
                    if (j == i) continue;
                    local_pool->check_and_update_mpi(local_pool->get_bod
y(i), local_pool->get_body(j), radius, gravity);
                }
            }
            for (size_t i = rank * chunk; i < i_max; ++i) {
                local_pool->get_body(i).update_for_tick(elapse, space,
radius);
            }
            // Send back the location and velocity
            MPI_Gather(&local_pool->x[chunk * rank], chunk, MPI_DOUBLE,
x, chunk, MPI_DOUBLE, 0, MPI_COMM_WORLD);
            MPI_Gather(&local_pool->y[chunk * rank], chunk, MPI_DOUBLE,
y, chunk, MPI_DOUBLE, 0, MPI_COMM_WORLD);
            MPI_Gather(&local_pool->vx[chunk * rank], chunk, MPI_DOUBLE,
vx, chunk, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

```
            MPI_Gather(&local_pool->vy[chunk * rank], chunk, MPI_DOUBLE,
vy, chunk, MPI_DOUBLE, 0, MPI_COMM_WORLD);
        }
    }

}
```

2. Pthread Version

```
#include <graphic/graphic.hpp>
#include <imgui_impl_sdl.h>
#include <cstring>
#include <nbody/body.hpp>
#include <mpi.h>
#include <chrono>
#include <iostream>
#include <iomanip>
#include <cstring>
#include <cstdint>
#include <cstddef>

template <typename ...Args>
void UNUSED(Args&&... args [[maybe_unused]]) {}

pthread_mutex_t* mutex;
pthread_barrier_t barrier;

float gravity = 100;
float space = 800;
float radius = 5;
int bodies = 20;
float elapse = 0.001;
ImVec4 color = ImVec4(1.0f, 1.0f, 0.4f, 1.0f);
float max_mass = 50;
int max_iteration = 100;
int chunk = 0;

BodyPool* global_pool;

void* gui_loop(void* t) {

    int* int_ptr = (int *) t;
```

```
    int tid = *int_ptr;
    int num_iteration = 0;

    // max index for each process
    int i_max = 0;
    if (chunk * (tid + 1) > (int) global_pool->size()) {
        i_max = (int) global_pool->size();
    }
    else {
        i_max = chunk * (tid + 1);
    }

    if (0 == tid) {
        size_t duration = 0;
        size_t bodies_count = 0;

        static float current_space = space;
        static float current_max_mass = max_mass;
        static int current_bodies = bodies;

        graphic::GraphicContext context{"Assignment 2"};
        context.run([&](graphic::GraphicContext *context
[[maybe_unused]], SDL_Window *) {
            auto io = ImGui::GetIO();
            ImGui::SetNextWindowPos(ImVec2(0.0f, 0.0f));
            ImGui::SetNextWindowSize(io.DisplaySize);
            ImGui::Begin("Assignment 2", nullptr,
                         ImGuiWindowFlags_NoMove
                         | ImGuiWindowFlags_NoCollapse
                         | ImGuiWindowFlags_NoTitleBar
                         | ImGuiWindowFlags_NoResize);
            ImDrawList *draw_list = ImGui::GetWindowDrawList();
            ImGui::Text("Application average %.3f ms/frame (%.1f FPS)",
1000.0f / ImGui::GetIO().Framerate,
                        ImGui::GetIO().Framerate);
            ImGui::DragFloat("Space", &current_space, 10, 200, 1600,
"%f");
            ImGui::DragFloat("Gravity", &gravity, 0.5, 0, 1000, "%f");
            ImGui::DragFloat("Radius", &radius, 0.5, 2, 20, "%f");
            ImGui::DragInt("Bodies", &current_bodies, 1, 2, 100, "%d");
            ImGui::DragFloat("Elapse", &elapse, 0.001, 0.001, 10, "%f");
            ImGui::DragFloat("Max Mass", &current_max_mass, 0.5, 5, 100,
"%f");
            ImGui::ColorEdit4("Color", &color.x);
```

```
            // Parameter adjustment in GUI is disabled
            auto begin = std::chrono::high_resolution_clock::now();
            if (num_iteration < max_iteration) {
                for (size_t i = tid * chunk; i < i_max; ++i) {
                    global_pool->ax[i] = 0;
                    global_pool->ay[i] = 0;
                    for (size_t j = 0; j < global_pool->size(); ++j) {
                        if (j == i) continue;
                        BodyPool::Body body_i = global_pool->get_body(i);
                        BodyPool::Body body_j = global_pool->get_body(j);
                        if (body_i.collide(body_j, radius))
{
                            int tid2 = (int) j / chunk;
                            if (tid2 < tid) {
                                pthread_mutex_lock(&mutex[tid2]);
                                pthread_mutex_lock(&mutex[tid]);
                            }
                            else if (tid2 > tid) {
                                pthread_mutex_lock(&mutex[tid]);
                                pthread_mutex_lock(&mutex[tid2]);
                            }
                            global_pool->check_and_update_pthread_collisi
on(body_i, body_j, radius, gravity);
                            pthread_mutex_unlock(&mutex[tid]);
                            if (tid2 != tid) {
                                pthread_mutex_unlock(&mutex[tid2]);
                            }
                        }
                        else {
                            global_pool->check_and_update_pthread_no_coll
ision(body_i, body_j, radius, gravity);
                        }
                    }
                }
                pthread_barrier_wait(&barrier);
                for (size_t i = tid * chunk; i < i_max; ++i) {
                    global_pool->get_body(i).update_for_tick(elapse,
space, radius);
                }
                num_iteration ++;
                bodies_count += bodies;
                pthread_barrier_wait(&barrier);
            }
```

```
            auto end = std::chrono::high_resolution_clock::now();
            duration += duration_cast<std::chrono::nanoseconds>(end -
begin).count();
            // only print the result once
            static bool isResultPrinted = false;
            if (!isResultPrinted && num_iteration >= max_iteration) {
                std::cout << bodies_count << " bodies in last " <<
duration << " nanoseconds\n";
                auto speed = static_cast<double>(bodies_count) /
static_cast<double>(duration) * 1e6;
                std::cout << "speed: " << std::setprecision(12) << speed
<< " bodies per millisecond" << std::endl;
                isResultPrinted = true;
            }
            // draw the balls
            {
                const ImVec2 p = ImGui::GetCursorScreenPos();
                for (size_t i = 0; i < global_pool->size(); ++i) {
                    auto body = global_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});
                }
            }

            ImGui::End();
        });
    }
    else {
        while (true) {
            for (size_t i = tid * chunk; i < i_max; ++i) {
                global_pool->ax[i] = 0;
                global_pool->ay[i] = 0;
                for (size_t j = 0; j < global_pool->size(); ++j) {
                    if (j == i) continue;
                    BodyPool::Body body_i = global_pool->get_body(i);
                    BodyPool::Body body_j = global_pool->get_body(j);
                    if (body_i.collide(body_j, radius)) {
                        int tid2 = (int) j / chunk;
                        if (tid2 < tid) {
                            pthread_mutex_lock(&mutex[tid2]);
                            pthread_mutex_lock(&mutex[tid]);
                        }
```

```
                        else if (tid2 > tid) {
                            pthread_mutex_lock(&mutex[tid]);
                            pthread_mutex_lock(&mutex[tid2]);
                        }
                        else {
                            pthread_mutex_lock(&mutex[tid]);
                        }
                        global_pool->check_and_update_pthread_collision(b
ody_i, body_j, radius, gravity);
                        pthread_mutex_unlock(&mutex[tid]);
                        if (tid2 != tid) {
                            pthread_mutex_unlock(&mutex[tid2]);
                        }
                    }
                    else {
                        global_pool->check_and_update_pthread_no_collisio
n(body_i, body_j, radius, gravity);
                    }
                }
            }
            pthread_barrier_wait(&barrier);
            for (size_t i = tid * chunk; i < i_max; ++i) {
                global_pool->get_body(i).update_for_tick(elapse, space,
radius);
            }
            pthread_barrier_wait(&barrier);
        }
    }
    return nullptr;
}


int main(int argc, char **argv) {

    if (argc < 2) {
        exit(1);
    }
    int num_threads = atoi(argv[1]);

    if (argc >= 3) {
        gravity = atof(argv[2]);
    }
    if (argc >= 4) {
        bodies = atof(argv[3]);
```

```
    }
    if (argc >= 5) {
        elapse = atof(argv[4]);
    }
    if (argc >= 6) {
        max_iteration = atoi(argv[5]);
    }

    chunk = (bodies + num_threads - 1) / num_threads;
    global_pool = new BodyPool(static_cast<size_t>(bodies),
static_cast<size_t>(chunk * num_threads), space, max_mass);

    mutex = new pthread_mutex_t[num_threads];
    for (int i = 0; i < num_threads; i++) {
        mutex[i] = PTHREAD_MUTEX_INITIALIZER;
    }

    pthread_barrier_init(&barrier, nullptr, num_threads);
    pthread_t threads[num_threads];
    int tids[num_threads];
    for (int i = 0; i < num_threads; i++) {
        tids[i] = i;
        void* tid = (void*) &tids[i];
        pthread_create(&threads[i], NULL, gui_loop, tid);
    }
    for (auto &i : threads) {
        pthread_join(i, nullptr);
    }
    return 0;
}
```

3. OpenMP Version

```
#include <graphic/graphic.hpp>
#include <imgui_impl_sdl.h>
#include <cstring>
#include <nbody/body.hpp>
#include <mpi.h>
#include <omp.h>
#include <chrono>
```

```cpp
#include <iostream>
#include <iomanip>
#include <cstring>
#include <cstdint>
#include <cstddef>

template <typename ...Args>
void UNUSED(Args&&... args [[maybe_unused]]) {}

omp_lock_t* lock;

int main(int argc, char **argv) {

    if (argc < 2) {
        exit(1);
    }
    int n_threads = atoi(argv[1]);

    static float gravity = 100;
    static float space = 800;
    static float radius = 5;
    static int bodies = 20;
    static float elapse = 0.001;
    ImVec4 color = ImVec4(1.0f, 1.0f, 0.4f, 1.0f);
    static float max_mass = 50;
    int max_iteration = 100;
    int chunk = 0;

    if (argc >= 3) {
        gravity = atof(argv[2]);
    }
    if (argc >= 4) {
        bodies = atof(argv[3]);
    }
    if (argc >= 5) {
        elapse = atof(argv[4]);
    }
    if (argc >= 6) {
        max_iteration = atoi(argv[5]);
    }

    lock = new omp_lock_t[n_threads];
    for (int i = 0; i < n_threads; i++) {
        omp_init_lock(&lock[i]);
```

```
    }

    chunk = (bodies + n_threads - 1) / n_threads;
    BodyPool bp(static_cast<size_t>(bodies), static_cast<size_t>(chunk *
n_threads), space, max_mass);
    BodyPool* global_pool = &bp;

    #pragma omp parallel num_threads(n_threads)
    {
        long tid = omp_get_thread_num();
        int num_iteration = 0;

        // max index for each process
        int i_max = 0;
        if (chunk * (tid + 1) > (int) global_pool->size()) {
            i_max = (int) global_pool->size();
        }
        else {
            i_max = chunk * (tid + 1);
        }

        if (0 == tid) {
            size_t duration = 0;
            size_t bodies_count = 0;

            static float current_space = space;
            static float current_max_mass = max_mass;
            static int current_bodies = bodies;

            graphic::GraphicContext context{"Assignment 2"};
            context.run([&](graphic::GraphicContext *context
[[maybe_unused]], SDL_Window *) {
                auto io = ImGui::GetIO();
                ImGui::SetNextWindowPos(ImVec2(0.0f, 0.0f));
                ImGui::SetNextWindowSize(io.DisplaySize);
                ImGui::Begin("Assignment 2", nullptr,
                             ImGuiWindowFlags_NoMove
                             | ImGuiWindowFlags_NoCollapse
                             | ImGuiWindowFlags_NoTitleBar
                             | ImGuiWindowFlags_NoResize);
                ImDrawList *draw_list = ImGui::GetWindowDrawList();
                ImGui::Text("Application average %.3f ms/frame (%.1f
FPS)", 1000.0f / ImGui::GetIO().Framerate,
                            ImGui::GetIO().Framerate);
```

```
                ImGui::DragFloat("Space", &current_space, 10, 200, 1600,
"%f");
                ImGui::DragFloat("Gravity", &gravity, 0.5, 0, 1000,
"%f");
                ImGui::DragFloat("Radius", &radius, 0.5, 2, 20, "%f");
                ImGui::DragInt("Bodies", &current_bodies, 1, 2, 100,
"%d");
                ImGui::DragFloat("Elapse", &elapse, 0.001, 0.001, 10,
"%f");
                ImGui::DragFloat("Max Mass", &current_max_mass, 0.5, 5,
100, "%f");
                ImGui::ColorEdit4("Color", &color.x);

                // Parameter adjustment in GUI is disabled
                auto begin = std::chrono::high_resolution_clock::now();
                if (num_iteration < max_iteration) {
                    for (size_t i = tid * chunk; i < i_max; ++i) {
                        global_pool->ax[i] = 0;
                        global_pool->ay[i] = 0;
                        for (size_t j = 0; j < global_pool->size(); ++j)
{
                            if (j == i) continue;
                            BodyPool::Body body_i =
global_pool->get_body(i);
                            BodyPool::Body body_j =
global_pool->get_body(j);
                            if (body_i.collide(body_j, radius)) {
                                int tid2 = (int) j / chunk;
                                if (tid2 < tid) {
                                    omp_set_lock(&lock[tid2]);
                                    omp_set_lock(&lock[tid]);
                                }
                                else if (tid2 > tid) {
                                    omp_set_lock(&lock[tid]);
                                    omp_set_lock(&lock[tid2]);
                                }
                                else {
                                    omp_set_lock(&lock[tid]);
                                }
                                global_pool->check_and_update_pthread_col
lision(body_i, body_j, radius, gravity);
                                omp_unset_lock(&lock[tid]);
                                if (tid2 != tid) {
                                    omp_unset_lock(&lock[tid2]);
```

```
                            }
                        }
                        else {
                            global_pool->check_and_update_pthread_no_
collision(body_i, body_j, radius, gravity);
                        }
                    }
                }
                for (size_t i = tid * chunk; i < i_max; ++i) {
                    global_pool->get_body(i).update_for_tick(elapse,
space, radius);
                }
                num_iteration ++;
                bodies_count += bodies;
                #pragma omp barrier
            }
            auto end = std::chrono::high_resolution_clock::now();
            duration += duration_cast<std::chrono::nanoseconds>(end
- begin).count();
            // only print the result once
            static bool isResultPrinted = false;
            if (!isResultPrinted && num_iteration >= max_iteration)
{
                std::cout << bodies_count << " bodies in last " <<
duration << " nanoseconds\n";
                auto speed = static_cast<double>(bodies_count) /
static_cast<double>(duration) * 1e6;
                std::cout << "speed: " << std::setprecision(12) <<
speed << " bodies per millisecond" << std::endl;
                isResultPrinted = true;
            }
            // draw the balls
            {
                const ImVec2 p = ImGui::GetCursorScreenPos();
                for (size_t i = 0; i < global_pool->size(); ++i) {
                    auto body = global_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});
                }
            }

            ImGui::End();
```

```
            });
        }
        else {
            while (true) {
                for (size_t i = tid * chunk; i < i_max; ++i) {
                    global_pool->ax[i] = 0;
                    global_pool->ay[i] = 0;
                    for (size_t j = 0; j < global_pool->size(); ++j) {
                        if (j == i) continue;
                        BodyPool::Body body_i = global_pool->get_body(i);
                        BodyPool::Body body_j = global_pool->get_body(j);
                        if (body_i.collide(body_j, radius)) {
                            int tid2 = (int) j / chunk;
                            if (tid2 < tid) {
                                omp_set_lock(&lock[tid2]);
                                omp_set_lock(&lock[tid]);
                            }
                            else if (tid2 > tid) {
                                omp_set_lock(&lock[tid]);
                                omp_set_lock(&lock[tid2]);
                            }
                            else {
                                omp_set_lock(&lock[tid]);
                            }
                            global_pool->check_and_update_pthread_collisi
on(body_i, body_j, radius, gravity);
                            omp_unset_lock(&lock[tid]);
                            if (tid2 != tid) {
                                omp_unset_lock(&lock[tid2]);
                            }
                        }
                        else {
                            global_pool->check_and_update_pthread_no_coll
ision(body_i, body_j, radius, gravity);
                        }

                    }
                }
                for (size_t i = tid * chunk; i < i_max; ++i) {
                    global_pool->get_body(i).update_for_tick(elapse,
space, radius);
                }
                #pragma omp barrier
            }
```

```
        }
    }

    return 0;
}
```

4. CUDA Version

```
#include <graphic/graphic.hpp>
#include <imgui_impl_sdl.h>
#include <cstring>
#include <nbody/body.hpp>
#include <mpi.h>
#include <chrono>
#include <iostream>
#include <iomanip>
#include <cstring>
#include <cstdint>
#include <cstddef>
#include <cuda.h>
#include <cuda_runtime.h>
#include <device_launch_parameters.h>
#include <inttypes.h>

template <typename ...Args>
void UNUSED(Args&&... args [[maybe_unused]]) {}

int bodies = 20;
ImVec4 color = ImVec4(1.0f, 1.0f, 0.4f, 1.0f);
float max_mass = 50;
int max_iteration = 100;

__global__ void calculate(BodyPool* cuda_pool,
                          float* gravity,
                          float* space,
                          float* radius,
                          float* elapse,
                          int* chunk,
                          int* lock) {

    int tid = blockIdx.x;
    int i_max = 0;
```

```
    if (*chunk * (tid + 1) > (int) cuda_pool->size()) {
        i_max = (int) cuda_pool->size();
    }
    else {
        i_max = *chunk * (tid + 1);
    }
    for (size_t i = tid * *chunk; i < i_max; ++i) {
        cuda_pool->ax[i] = 0;
        cuda_pool->ay[i] = 0;
        for (size_t j = 0; j < cuda_pool->size(); ++j) {
            if (j == i) continue;
            BodyPool::Body body_i = cuda_pool->get_body(i);
            BodyPool::Body body_j = cuda_pool->get_body(j);
            if (body_i.collide(body_j, *radius)) {
                int tid2 = (int) j / *chunk;
                if (tid2 < tid) {
                    while (lock[tid2] != 0) {};
                    lock[tid2] = 1;
                    while (lock[tid] != 0) {};
                    lock[tid] = 1;
                }
                else if (tid2 > tid) {
                    while (lock[tid] != 0) {};
                    lock[tid] = 1;
                    while (lock[tid2] != 0) {};
                    lock[tid2] = 1;
                }
                else {
                    while (lock[tid] != 0) {};
                    lock[tid] = 1;
                }
                cuda_pool->check_and_update_pthread_collision(body_i,
body_j, *radius, *gravity);
                lock[tid] = 0;
                if (tid2 != tid) {
                    lock[tid2] = 0;
                }
            }
            else {
                cuda_pool->check_and_update_pthread_no_collision(body_i,
body_j, *radius, *gravity);
            }
        }
    }
```

```
    __syncthreads();
    for (size_t i = tid * *chunk; i < i_max; ++i) {
        cuda_pool->get_body(i).update_for_tick(*elapse, *space,
*radius);
    }
    __syncthreads();
}

int main(int argc, char **argv) {

    if (argc < 2) {
        exit(1);
    }
    int num_threads = atoi(argv[1]);

    float host_gravity = 100;
    float host_space = 800;
    float host_radius = 5;
    float host_elapse = 0.001;

    if (argc >= 3) {
        host_gravity = atof(argv[2]);
    }
    if (argc >= 4) {
        bodies = atof(argv[3]);
    }
    if (argc >= 5) {
        host_elapse = atof(argv[4]);
    }
    if (argc >= 6) {
        max_iteration = atoi(argv[5]);
    }

    int host_chunk = (bodies + num_threads - 1) / num_threads;

    BodyPool* cuda_pool;
    double* x;
    double* y;
    double* vx;
    double* vy;
    double* ax;
    double* ay;
    double* m;
    float* gravity;
```

```
    float* space;
    float* radius;
    float* elapse;
    int* chunk;
    int* lock;

    BodyPool bp(static_cast<size_t>(bodies),
static_cast<size_t>(host_chunk * num_threads), host_space, max_mass);

    cudaMallocManaged((void**) &cuda_pool, sizeof(bp));
    cudaMemcpy(cuda_pool, &bp, sizeof(bp), cudaMemcpyHostToDevice);
    cudaMallocManaged((void**) &gravity, sizeof(float));
    cudaMemcpy(gravity, &host_gravity, sizeof(float),
cudaMemcpyHostToDevice);
    cudaMallocManaged((void**) &space, sizeof(float));
    cudaMemcpy(space, &host_space, sizeof(float),
cudaMemcpyHostToDevice);
    cudaMallocManaged((void**) &radius, sizeof(float));
    cudaMemcpy(radius, &host_radius, sizeof(float),
cudaMemcpyHostToDevice);
    cudaMallocManaged((void**) &elapse, sizeof(float));
    cudaMemcpy(elapse, &host_elapse, sizeof(float),
cudaMemcpyHostToDevice);
    cudaMallocManaged((void**) &chunk, sizeof(int));
    cudaMemcpy(chunk, &host_chunk, sizeof(int), cudaMemcpyHostToDevice);

    size_t array_size = sizeof(double) * host_chunk * num_threads;

    cudaMallocManaged((void**) &x, array_size);
    cudaMallocManaged((void**) &y, array_size);
    cudaMallocManaged((void**) &vx, array_size);
    cudaMallocManaged((void**) &vy, array_size);
    cudaMallocManaged((void**) &ax, array_size);
    cudaMallocManaged((void**) &ay, array_size);
    cudaMallocManaged((void**) &m, array_size);
    cuda_pool->x = x;
    cuda_pool->y = y;
    cuda_pool->vx = vx;
    cuda_pool->vy = vy;
    cuda_pool->ax = ax;
    cuda_pool->ay = ay;
    cuda_pool->m = m;
    cudaMemcpy(cuda_pool->m, bp.m, array_size, cudaMemcpyHostToDevice);
    cudaMemcpy(cuda_pool->x, bp.x, array_size, cudaMemcpyHostToDevice);
```

```
    cudaMemcpy(cuda_pool->y, bp.y, array_size, cudaMemcpyHostToDevice);
    cudaMemcpy(cuda_pool->vx, bp.vx, array_size,
cudaMemcpyHostToDevice);
    cudaMemcpy(cuda_pool->vy, bp.vy, array_size,
cudaMemcpyHostToDevice);

    cudaMallocManaged((void**) &lock, sizeof(int) * num_threads);
    cudaMemset(lock, 0, sizeof(int) * num_threads);

    cudaError_t cudaStatus;
    size_t duration = 0;
    size_t bodies_count = 0;

    static float current_space = host_space;
    static float current_max_mass = max_mass;
    static int current_bodies = bodies;
    int num_iteration = 0;

    cudaStatus = cudaGetLastError();
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "mykernel launch failed: %s\n",
                cudaGetErrorString(cudaStatus));
        return 0;
    }

    graphic::GraphicContext context{"Assignment 2"};
    context.run([&](graphic::GraphicContext *context [[maybe_unused]],
SDL_Window *) {
        auto io = ImGui::GetIO();
        ImGui::SetNextWindowPos(ImVec2(0.0f, 0.0f));
        ImGui::SetNextWindowSize(io.DisplaySize);
        ImGui::Begin("Assignment 2", nullptr,
                     ImGuiWindowFlags_NoMove
                     | ImGuiWindowFlags_NoCollapse
                     | ImGuiWindowFlags_NoTitleBar
                     | ImGuiWindowFlags_NoResize);
        ImDrawList *draw_list = ImGui::GetWindowDrawList();
        ImGui::Text("Application average %.3f ms/frame (%.1f FPS)",
1000.0f / ImGui::GetIO().Framerate,
                     ImGui::GetIO().Framerate);
        ImGui::DragFloat("Space", &current_space, 10, 200, 1600, "%f");
        ImGui::DragFloat("Gravity", &host_gravity, 0.5, 0, 1000, "%f");
        ImGui::DragFloat("Radius", &host_radius, 0.5, 2, 20, "%f");
        ImGui::DragInt("Bodies", &current_bodies, 1, 2, 100, "%d");
```

```
        ImGui::DragFloat("Elapse", &host_elapse, 0.001, 0.001, 10,
"%f");
        ImGui::DragFloat("Max Mass", &current_max_mass, 0.5, 5, 100,
"%f");
        ImGui::ColorEdit4("Color", &color.x);

        // Parameter adjustment in GUI is disabled
        auto begin = std::chrono::high_resolution_clock::now();
        if (num_iteration < max_iteration) {
            // Launch to GPU kernel
            calculate<<<1, num_threads>>>(cuda_pool, gravity, space,
radius, elapse, chunk, lock);
            cudaDeviceSynchronize();
            // Receive data from device
            cudaMemcpy(bp.x, cuda_pool->x, array_size,
cudaMemcpyDeviceToHost);
            cudaMemcpy(bp.y, cuda_pool->y, array_size,
cudaMemcpyDeviceToHost);
            cudaMemcpy(bp.vx, cuda_pool->vx, array_size,
cudaMemcpyDeviceToHost);
            cudaMemcpy(bp.vy, cuda_pool->vy, array_size,
cudaMemcpyDeviceToHost);

            cudaStatus = cudaGetLastError();
            if (cudaStatus != cudaSuccess) {
                fprintf(stderr, "mykernel launch failed: %s\n",
                        cudaGetErrorString(cudaStatus));
                return 0;
            }
            num_iteration ++;
            bodies_count += bodies;
        }
        auto end = std::chrono::high_resolution_clock::now();
        duration +=
std::chrono::duration_cast<std::chrono::nanoseconds>(end -
begin).count();
        // only print the result once
        static bool isResultPrinted = false;
        if (!isResultPrinted && num_iteration >= max_iteration) {
            std::cout << bodies_count << " bodies in last " << duration
<< " nanoseconds\n";
            auto speed = static_cast<double>(bodies_count) /
static_cast<double>(duration) * 1e6;
```

```
            std::cout << "speed: " << std::setprecision(12) << speed <<
" bodies per millisecond" << std::endl;
            isResultPrinted = true;
        }
        // draw the balls
        {
            const ImVec2 p = ImGui::GetCursorScreenPos();
            for (size_t i = 0; i < bodies; ++i) {
                auto x = p.x + static_cast<float>(bp.x[i]);
                auto y = p.y + static_cast<float>(bp.y[i]);
                draw_list->AddCircleFilled(ImVec2(x, y), host_radius,
ImColor{color});
            }
        }

        ImGui::End();
    });

    cudaDeviceReset();

    return 0;
}
```