Assignment Report

CSC 4005 Heat Distribution

Wei Wu (吴畏)

118010335

December 5, 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 pthread program, an CUDA program, an OpenMP program and an MPI program to simulate an two-dimensional heat distribution problem.

A square metal sheet has known temperatures along each of its edges. Find the temperature distribution.

Dividing the area into a fine mesh of points, $h_{i,j}$. The temperature at an inside point can be taken to be the average of the temperatures of the four neighboring points.

Convenient to describe the edges by points adjacent to the interior points. The interior points of $h_{i,j}$ are where $0 < i < n$, $0 < j < n$ [$(n-1) \times (n-1)$ interior points].

Temperature of each point by iterating the equation:

$$h_{i,j} = \frac{h_{i-1,j} + h_{i+1,j} + h_{i,j-1} + h_{i,j+1}}{4}$$

($0 < i < n$, $0 < j < n$) for a fixed number of iterations or until the difference between iterations of a point is less than some very small prescribed amount.

II. Design Approaches

1. Architecture

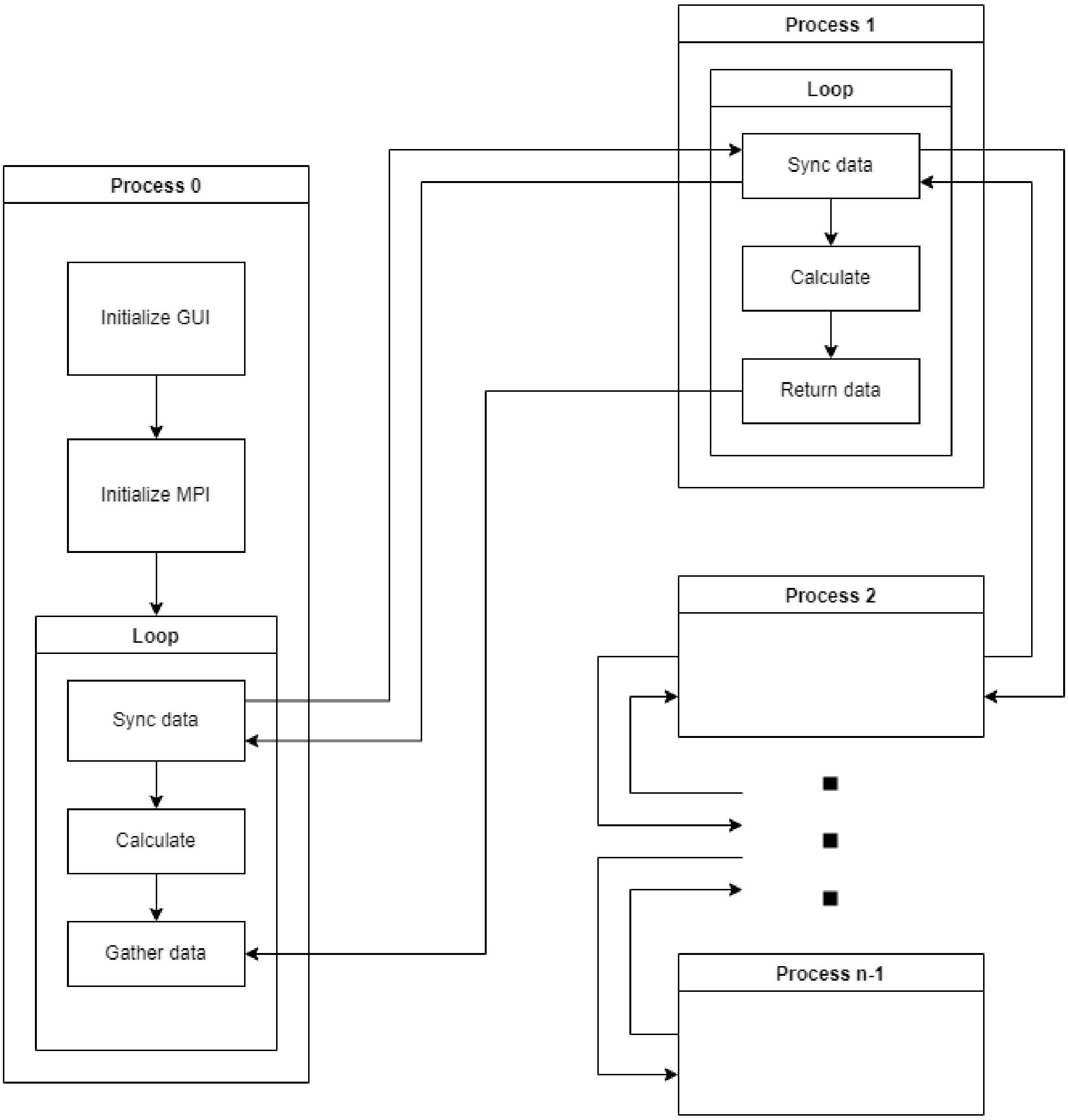

Figure 1. Architecture of the MPI version of the program

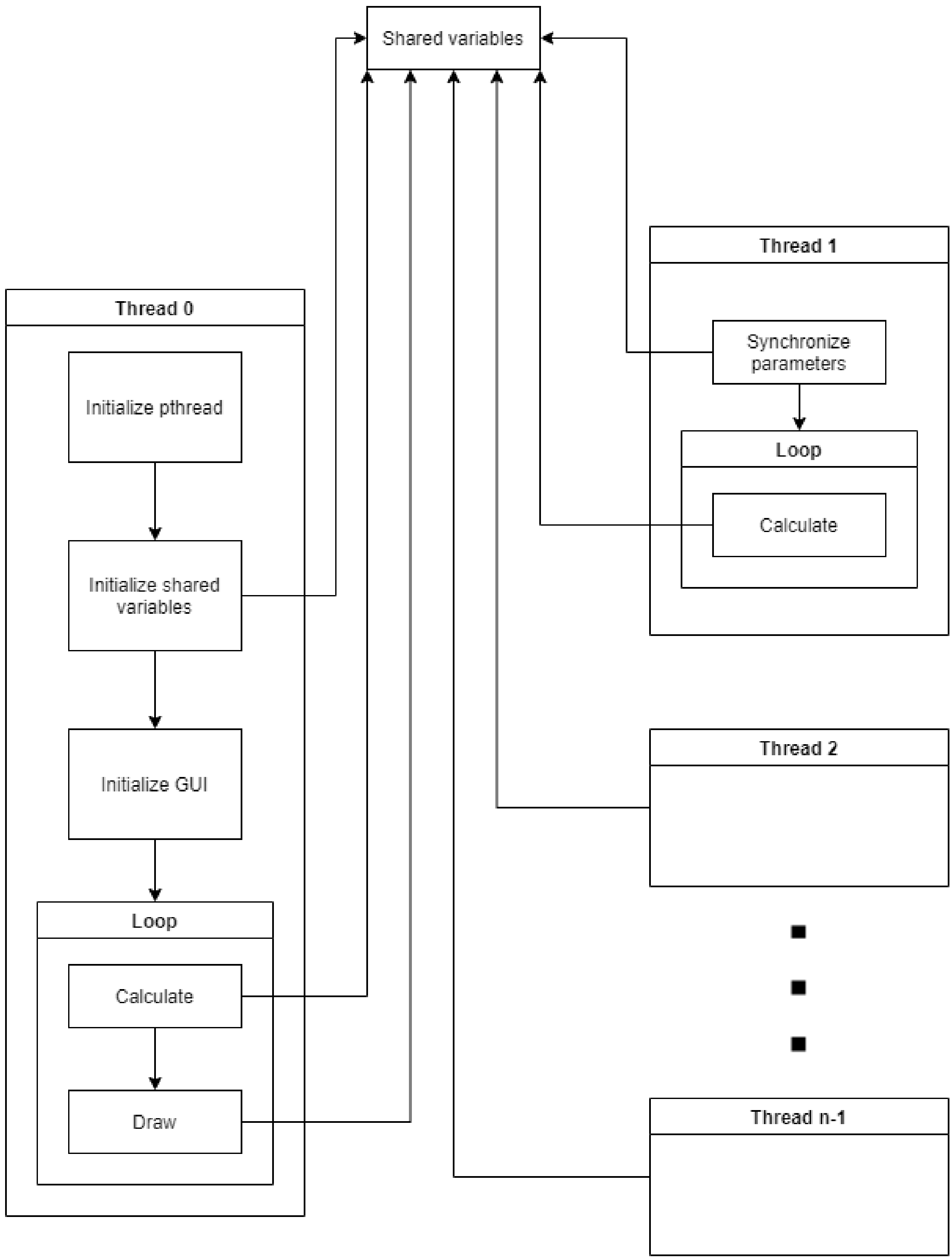

Figure 2. Architecture of the pthread version of the program

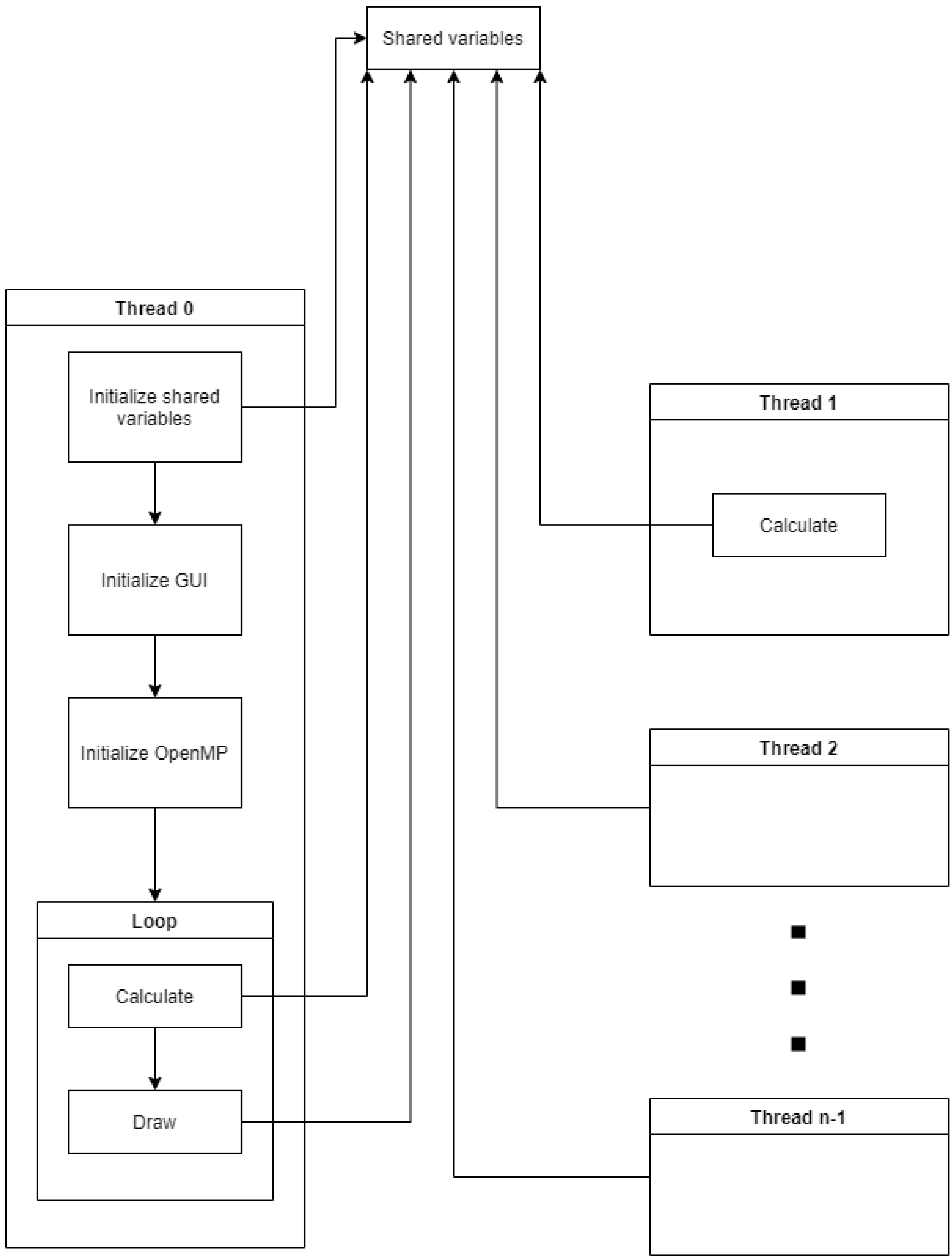

Figure 3. Architecture of the OpenMP version of the program

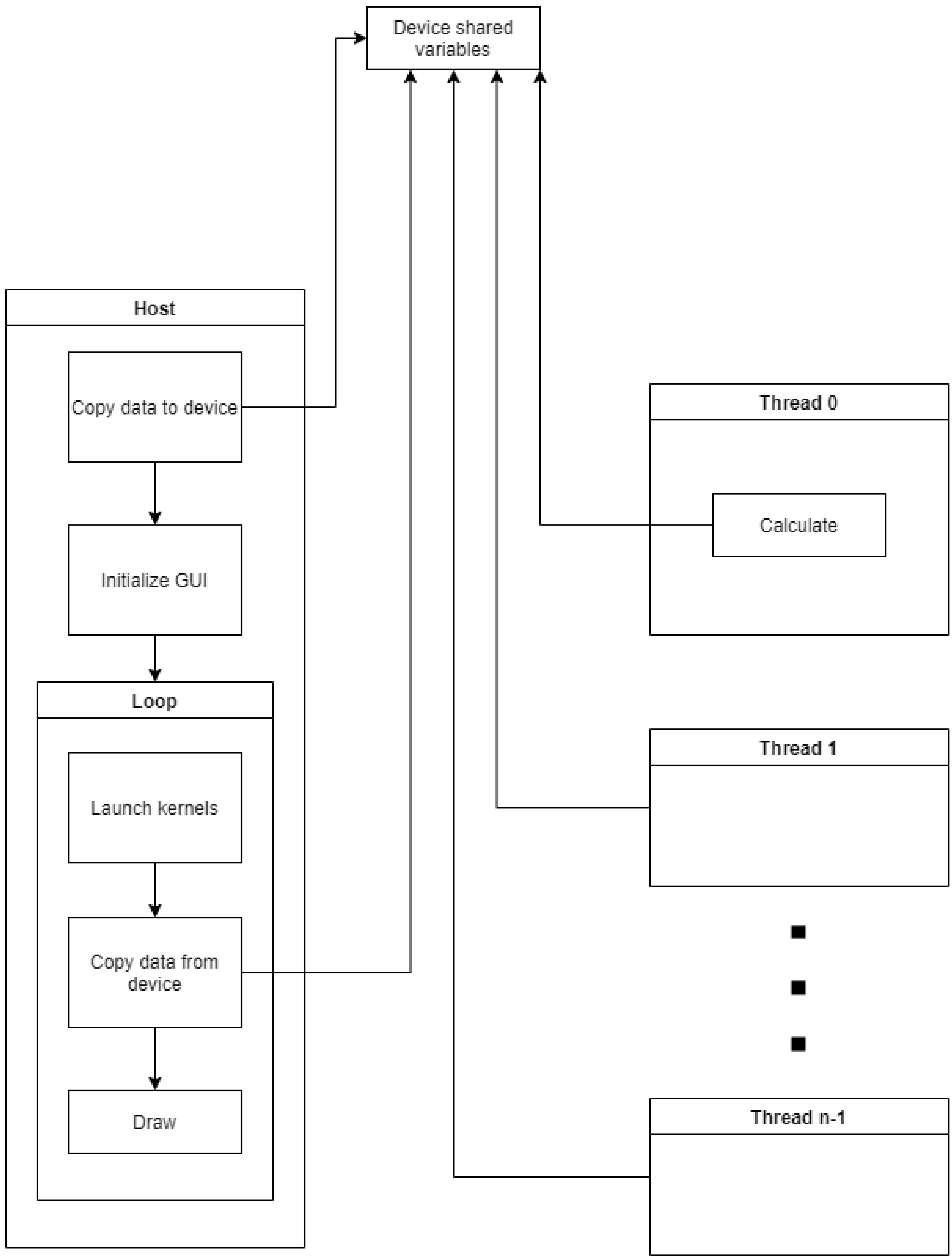

Figure 4. Architecture of the CUDA version of the program

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 (room_size) (max_iteration) (algorithm)

Examples:

a. mpirun csc4005_imgui

b. mpirun csc4005_imgui 300 100 0

c. mpirun csc4005_imgui 300 100 1

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 (room_size) (max_iteration) (algorithm)

Examples:

a. srun -n1 csc4005_imgui 4

b. srun -n1 csc4005_imgui 8 100 100

c. srun -n1 csc4005_imgui 16 1000 100 1

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 (room_size) (max_iteration) (algorithm)

Examples:

a. srun -n1 csc4005_imgui 4

b. srun -n1 csc4005_imgui 8 100 100 0

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 (room_size) (max_iteration) (algorithm)

Examples:

c. srun -n1 csc4005_imgui 4

d. srun -n1 csc4005_imgui 32 1000 100 1

IV. Performance Analysis

1. Test Results

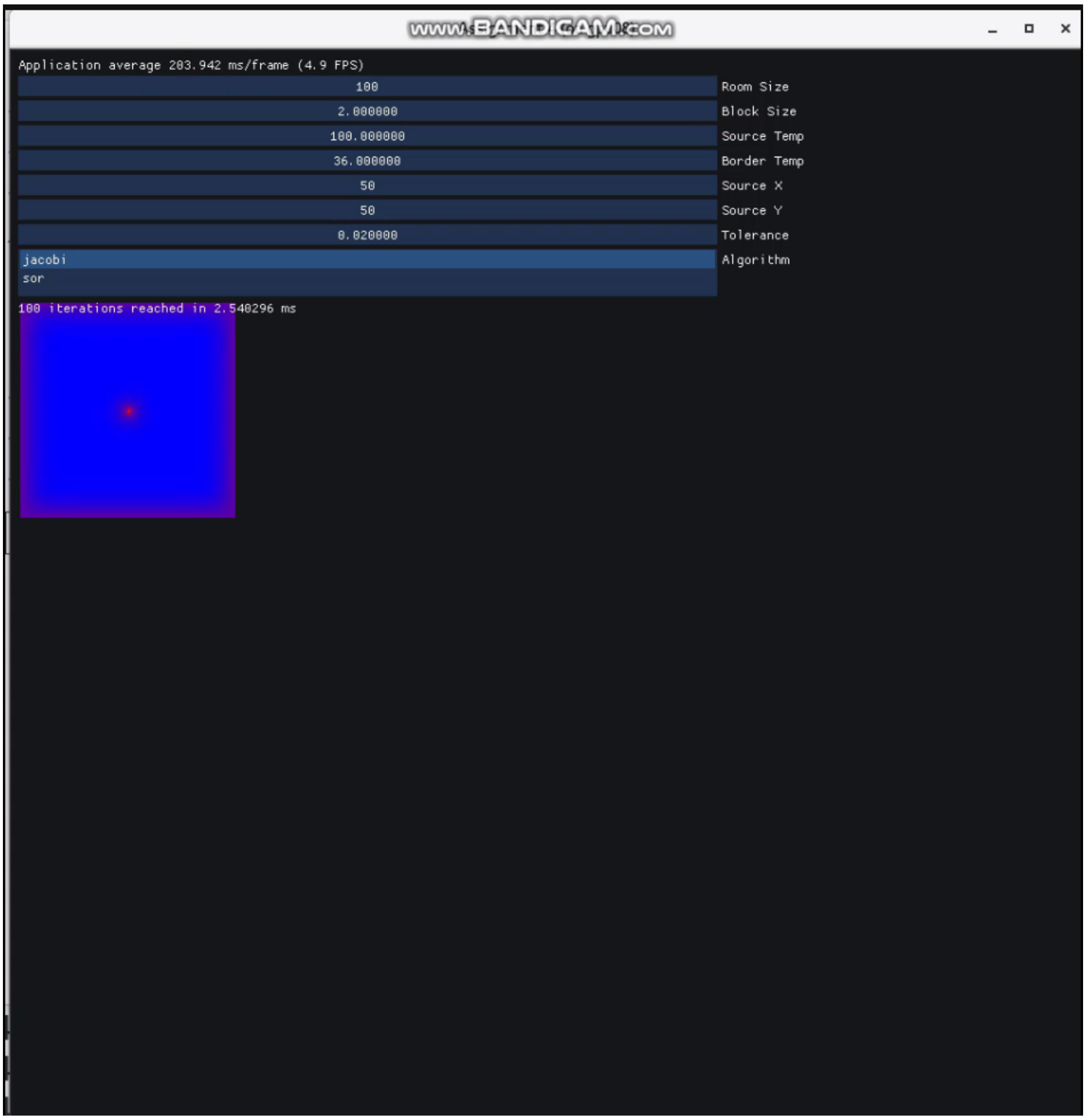

Figure 6. Test result with room_size = 100

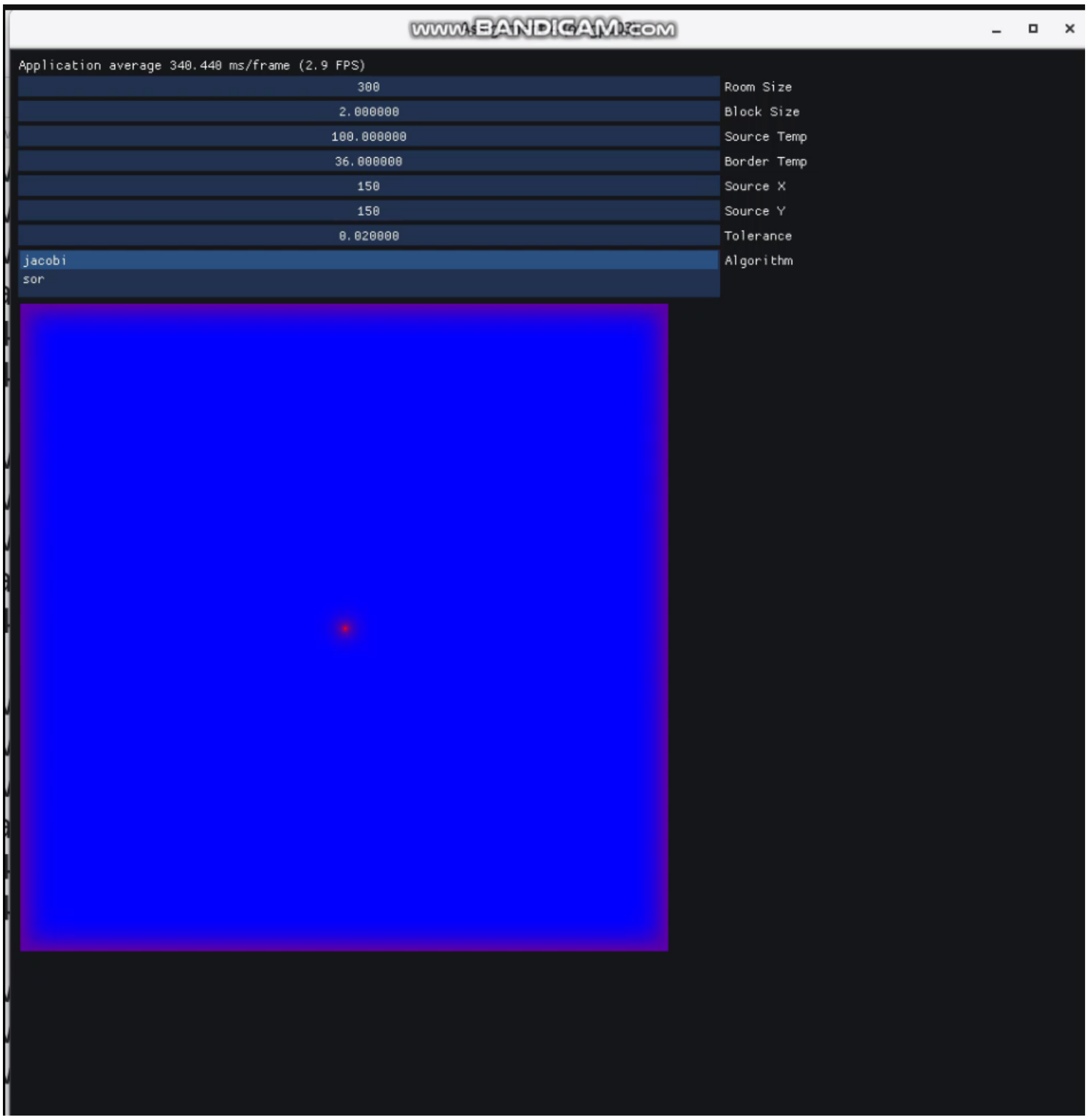

Figure 7. Test result with room_size = 300

Figure 8. Test result with room_size = 1000

room size	Number of processes				
	1	4	16	32	64
100	8.48	4.01	2.56	2.54	69.92
300	75.35	21.89	7.67	6.13	38.66
1000	815.43	208.98	60.69	38.54	44.35

Table 1. Execution time (ms) of 100 iterations using the Jacobi algorithm of the MPI version of the program with respect to different room sizes and numbers of processes

room size		Number of processes				
		1	4	16	32	64
	100	11.99	5.59	3.29	3.33	84.28
	300	107.8	30.45	10.03	7.66	57.18
	1000	1166.23	304.86	82.33	54.22	58.33

Table 2. Execution time (ms) of 100 iterations using the Sor algorithm of the MPI version of the program with respect to different room sizes and numbers of processes

room size		Number of processes				
		1	4	16	32	64
	100	8.92	6.44	15.88	28.51	59.87
	300	82.34	25.21	21.23	35.19	63.24
	1000	842.19	219.42	73.74	67.94	96.13

Table 3. Execution time (ms) of 100 iterations using the Jacobi algorithm of the pthread version of the program with respect to different room sizes and numbers of processes

room size		Number of processes				
		1	4	16	32	64
	100	13.32	10.11	32.71	68.95	140.7
	300	117.29	39.68	37.55	71.52	132.93
	1000	1257.33	336.49	115.53	125.17	195.75

Table 4. Execution time (ms) of 100 iterations using the Sor algorithm of the pthread version of the program with respect to different room sizes and numbers of processes

room size		Number of processes				
		1	4	16	32	64
	100	8.86	4.44	4.4	113.12	797.24
	300	79.49	23.27	9.65	69.99	1090.05
	1000	841.12	222.48	62.97	47.57	807.02

Table 5. Execution time (ms) of 100 iterations using the Jacobi algorithm of the OpenMP version of the program with respect to different room sizes and numbers of processes

		Number of processes				
		1	4	16	32	64
room size	100	14.09	6.89	7.49	117.46	892.84
	300	118.34	35.03	15.9	119.22	739.98
	1000	1279.02	329.07	104.59	164.05	1229.78

Table 6. Execution time (ms) of 100 iterations using the Sor algorithm of the OpenMP version of the program with respect to different room sizes and numbers of processes

		Number of processes				
		1	4	16	32	64
room size	100	535.43	142.86	49.75	33.95	22.35
	300	4802.93	1240.65	346.13	219.93	122.63
	1000	39326.86	10477.32	3870.53	2285.29	1292.92

Table 7. Execution time (ms) of 100 iterations using the Jacobi algorithm of the CUDA version of the program with respect to different room sizes and numbers of processes

		1	4	16	32	64
room size	100	988.32	372.58	142.99	52.49	29.52
	300	8023.61	4269.45	1136.82	363.76	339.43
	1000	74323.37	19468.78	9386.17	4005.14	2200.03

Table 8. Execution time (ms) of 100 iterations using the Sor algorithm of the CUDA version of the program with respect to different room sizes 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.

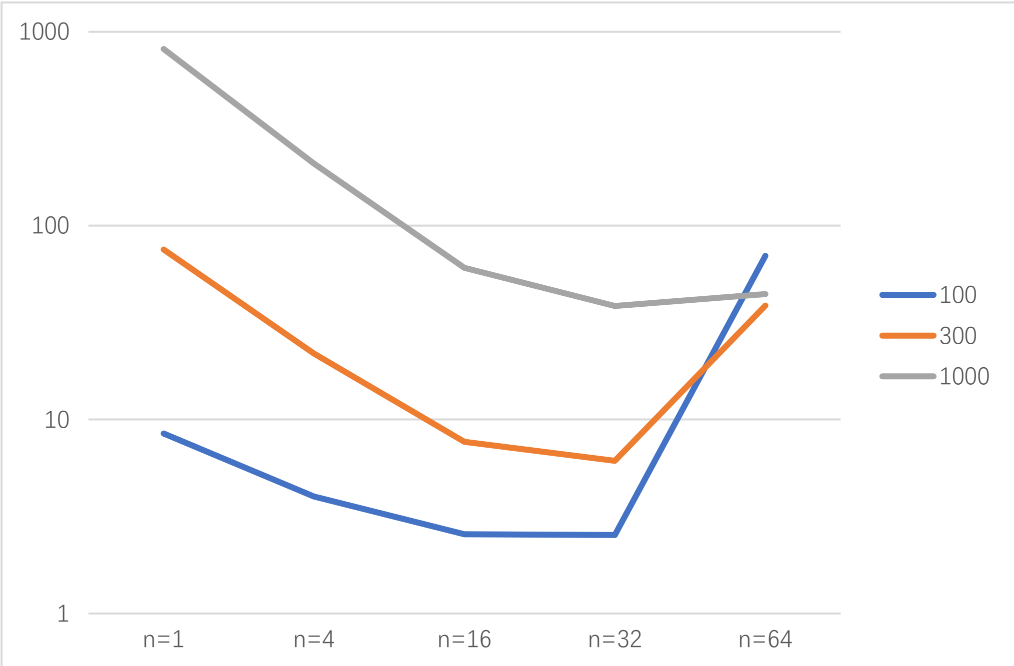

Figure 9. Execution time (ms) of 100 iterations using the Jacobi algorithm of the MPI version of the program with respect to different number of processes

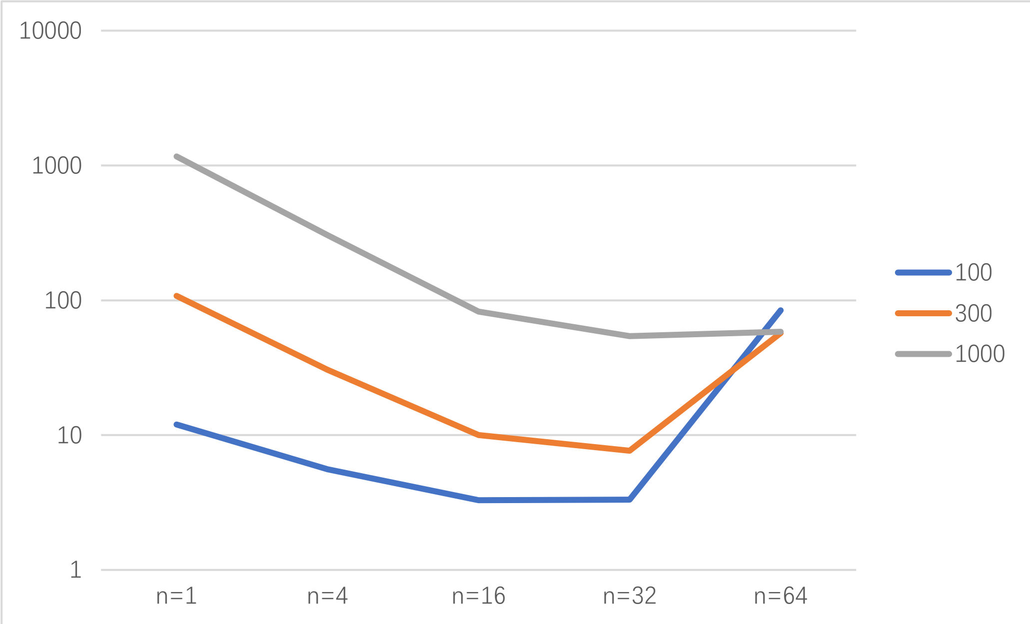

Figure 10. Execution time (ms) of 100 iterations using the Sor algorithm of the MPI version of the program with respect to different number of processes

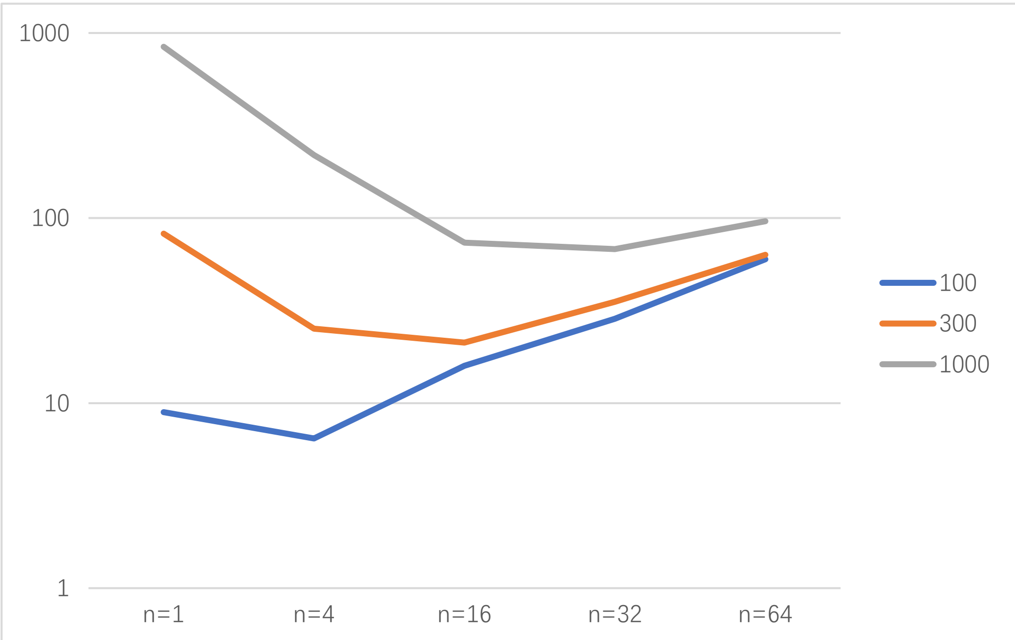

Figure 11. Execution time (ms) of 100 iterations using the Jacobi algorithm of the pthread version of the program with respect to different number of processes

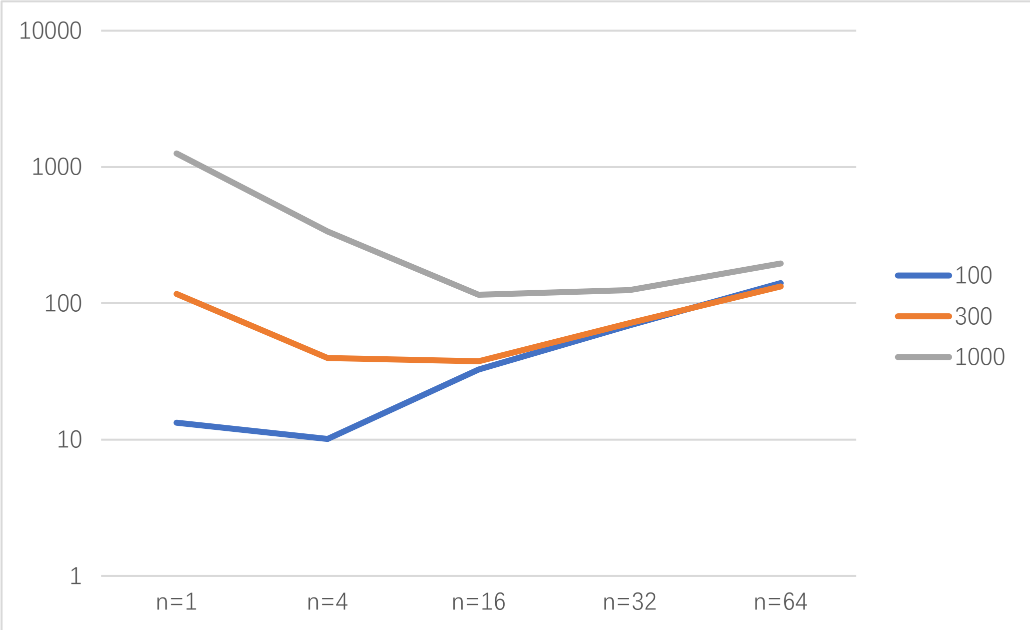

Figure 12. Execution time (ms) of 100 iterations using the Sor algorithm of the pthread version of the program with respect to different number of processes

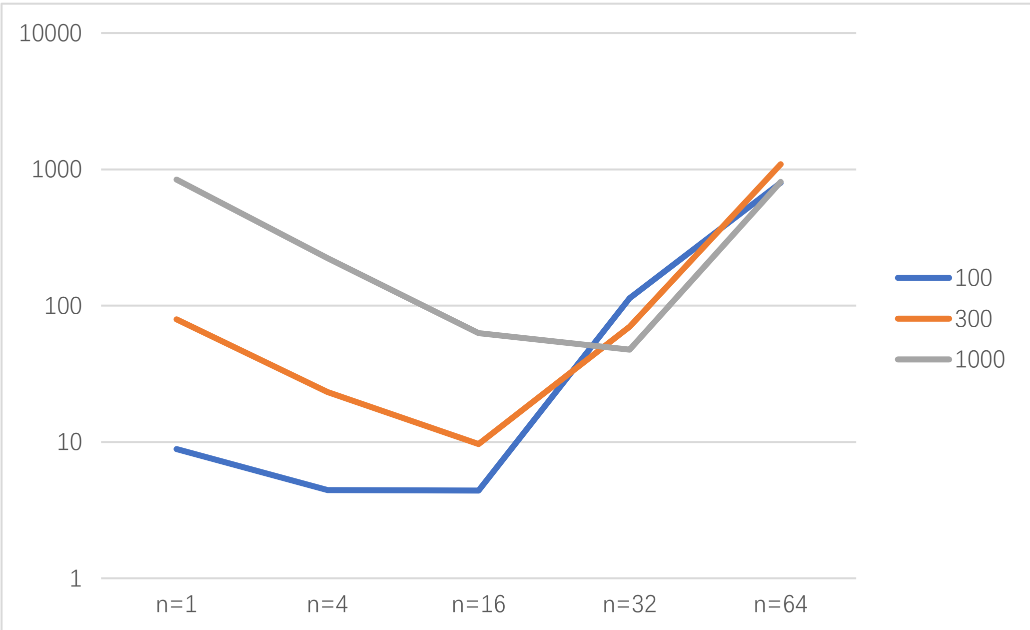

Figure 13. Execution time (ms) of 100 iterations using the Jacobi algorithm of the OpenMP version of the program with respect to different number of processes

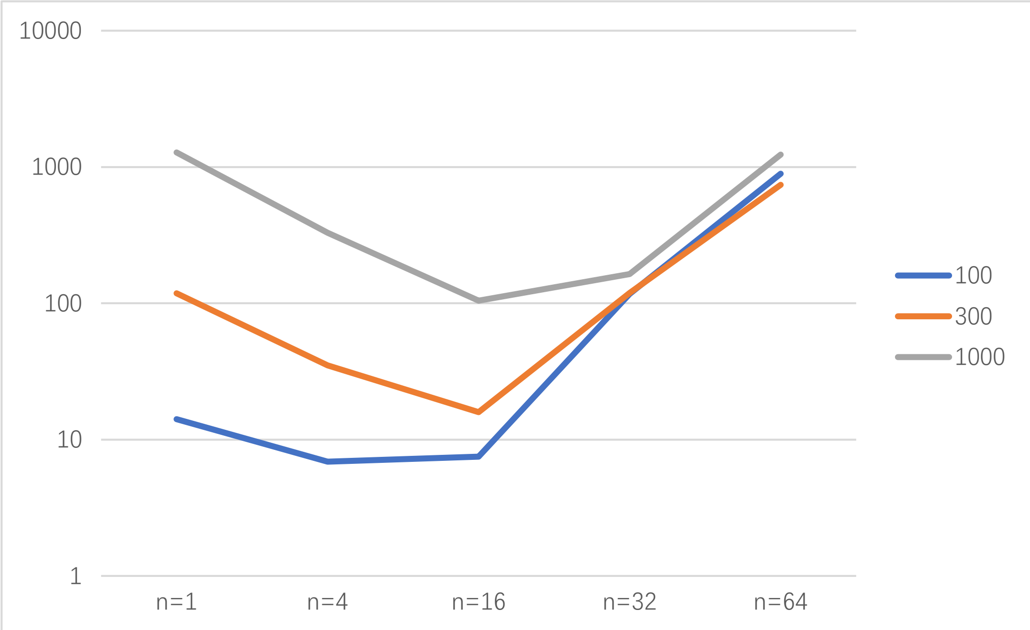

Figure 14. Execution time (ms) of 100 iterations using the Sor algorithm of the OpenMP version of the program with respect to different number of processes

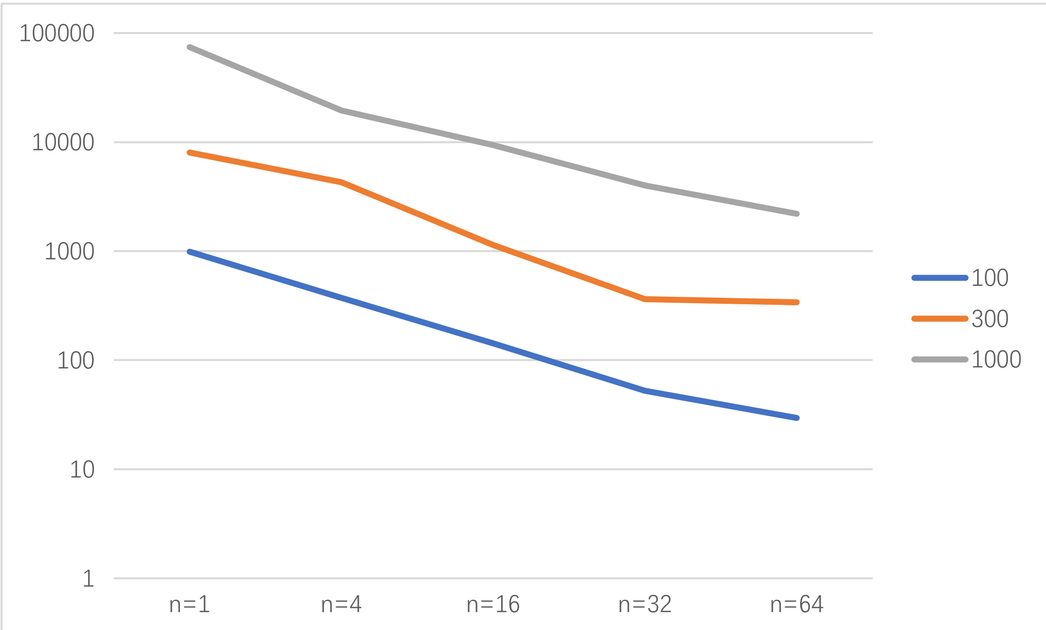

Figure 15. Execution time (ms) of 100 iterations using the Jacobi algorithm of the CUDA version of the program with respect to different number of processes

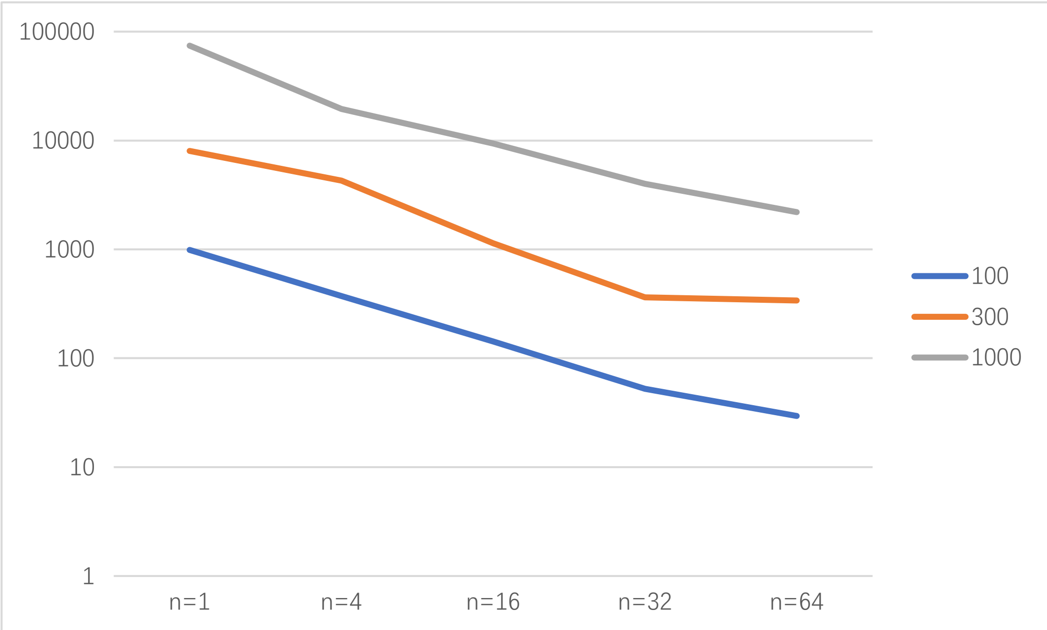

Figure 16. Execution time (ms) of 100 iterations using the Sor algorithm of the CUDA version of the program with respect to different number of processes

2. Analysis

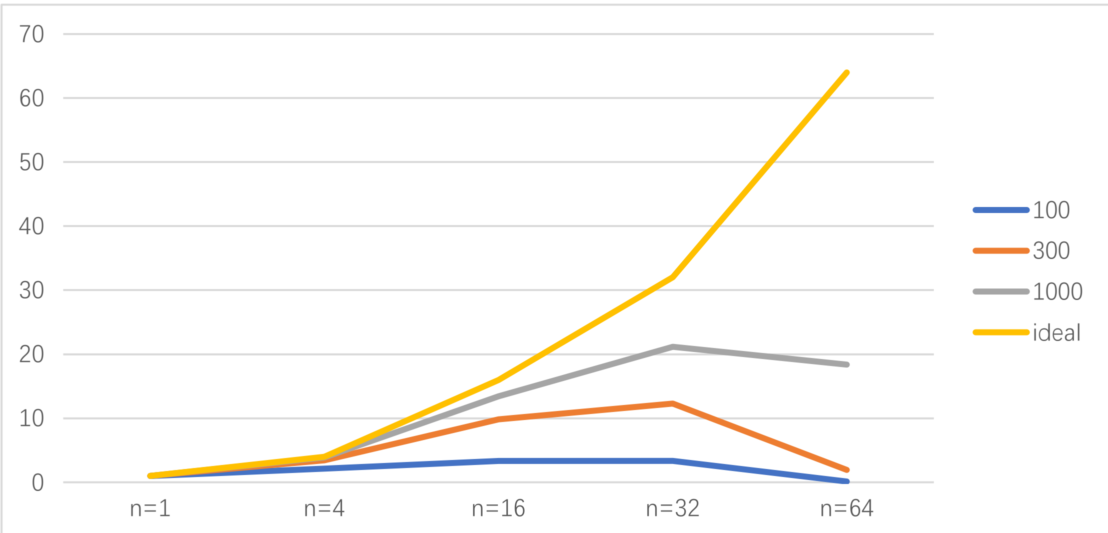

Figure 17. Speedup factor of the MPI version of the program using the Jacobi algorithm with respect to different number of processes

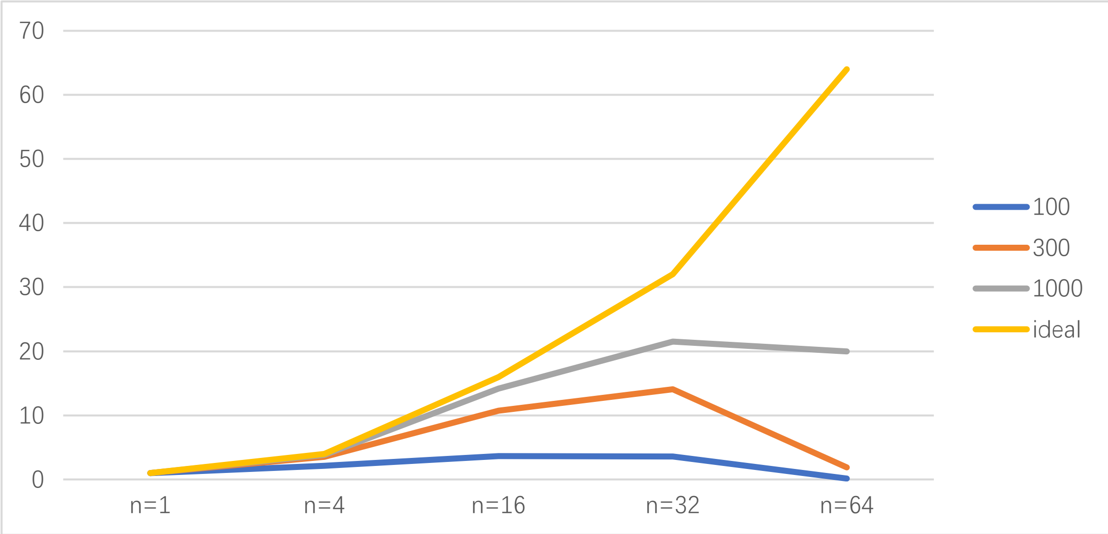

Figure 18. Speedup factor of the MPI version of the program using the Sor algorithm with respect to different number of processes

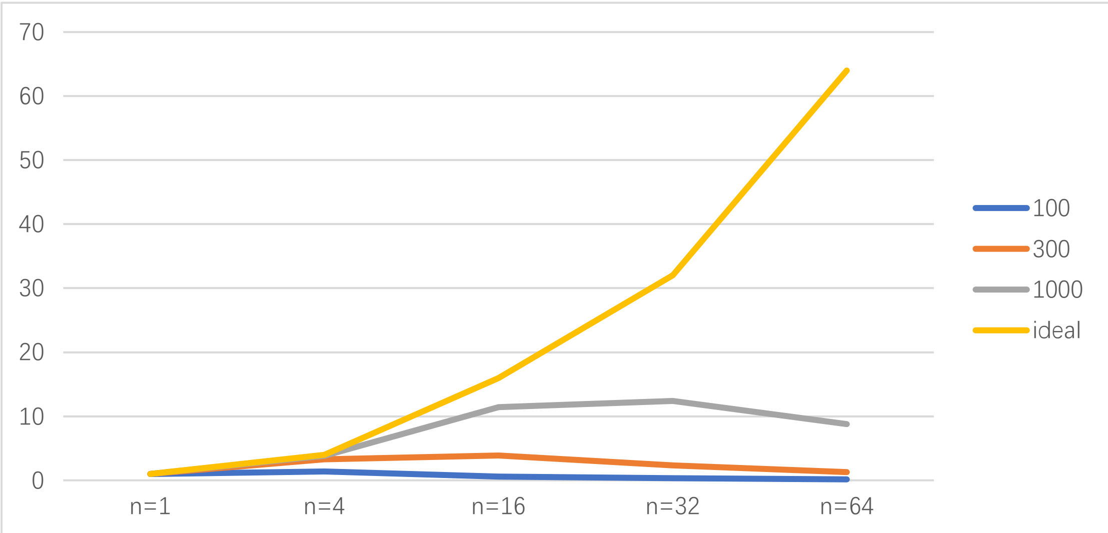

Figure 19. Speedup factor of the pthread version of the program using the Jacobi algorithm with respect to different number of processes

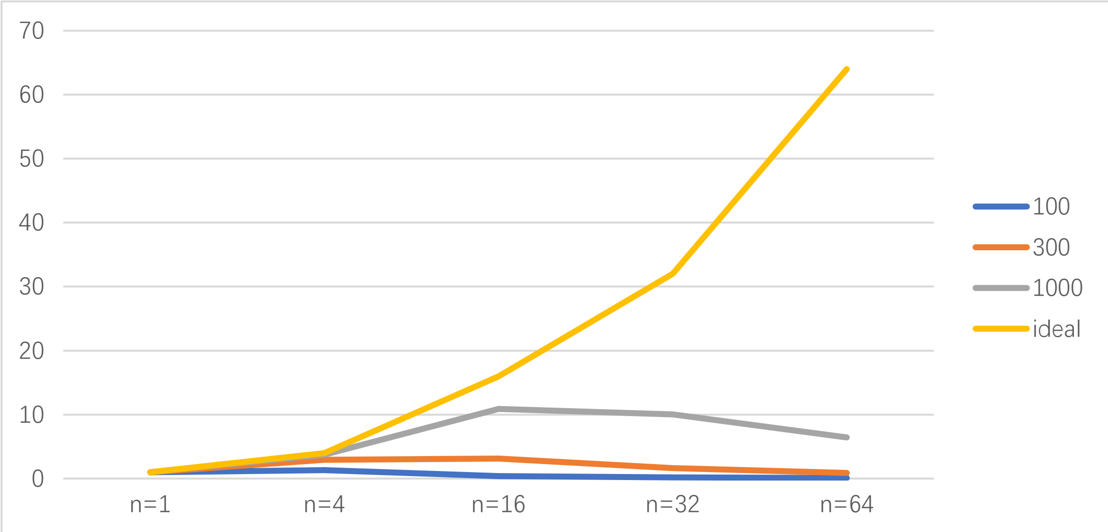

Figure 20. Speedup factor of the pthread version of the program using the Sor algorithm with respect to different number of processes

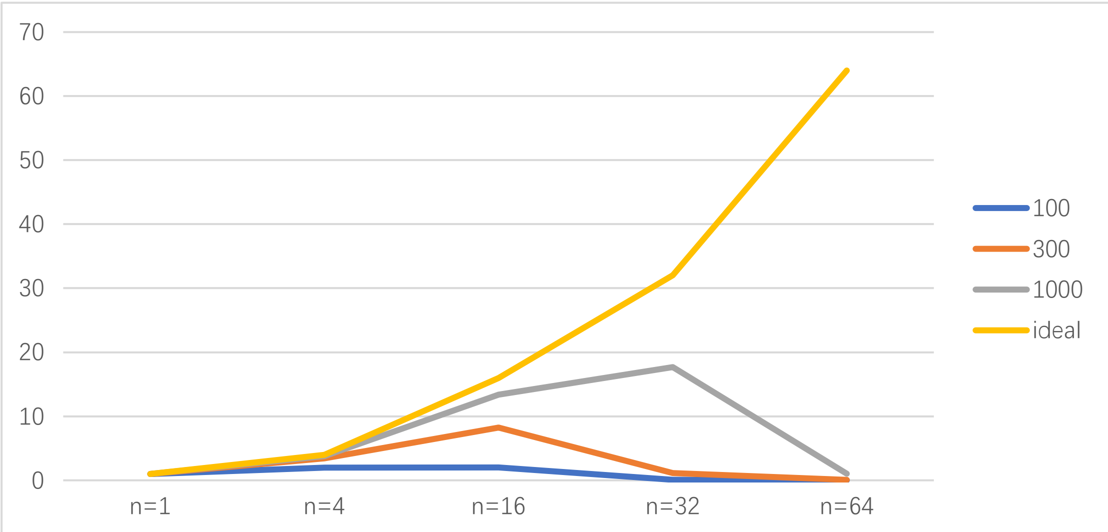

Figure 21. Speedup factor of the OpenMP version of the program using the Jacobi algorithm with respect to different number of processes

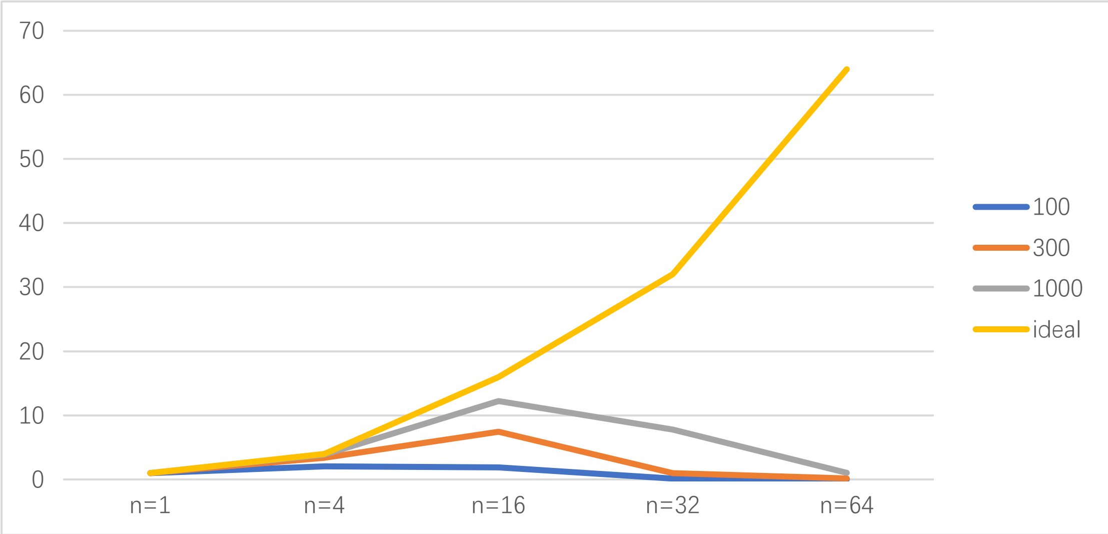

Figure 22. Speedup factor of the OpenMP version of the program using the Sor algorithm with respect to different number of processes

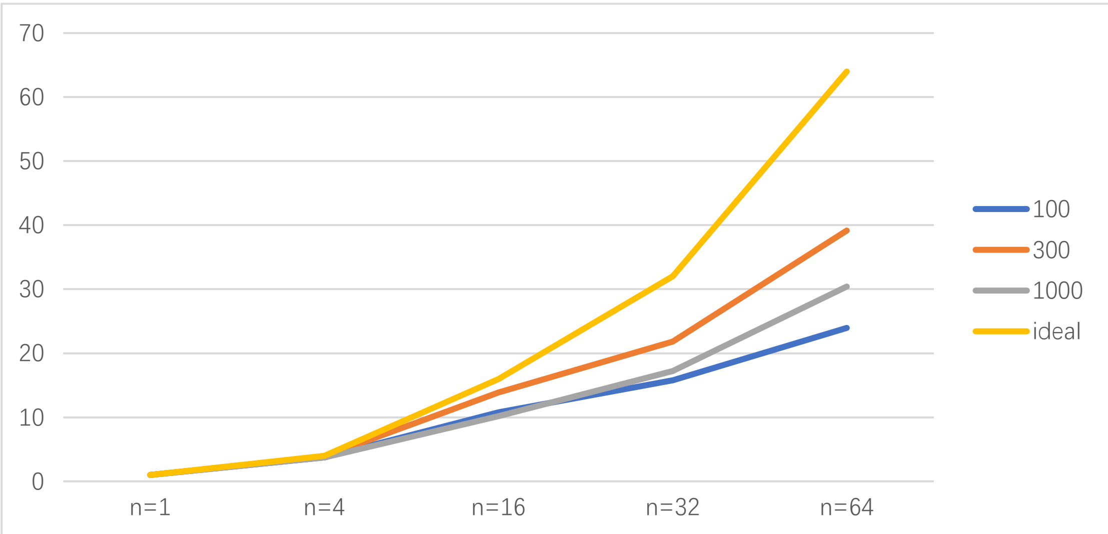

Figure 23. Speedup factor of the CUDA version of the program using the Jacobi algorithm with respect to different number of processes

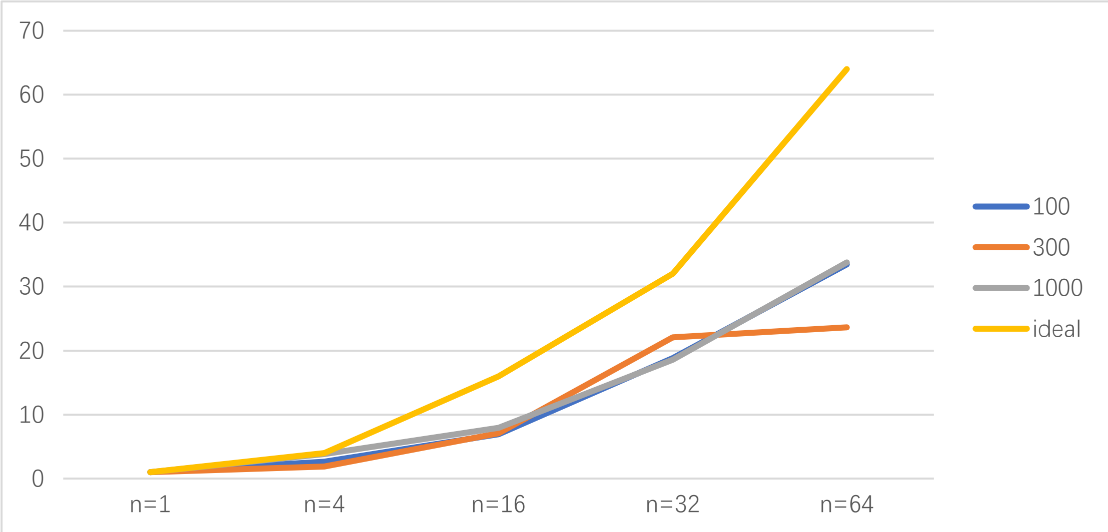

Figure 24. Speedup factor of the CUDA version of the program using the Sor algorithm with respect to different number of processes

As shown in Figure 17-24, when the room size is small (100) and the number of processes/threads is also small, the execution time does not decrease much as the number of processes/threads increases. When the number of processes/threads becomes larger, the execution time even soars 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 room sizes. For MPI, pthread, OpenMP versions, when the number of processes/threads exceeds 32, which is the number of processor cores on a single node, the performance drops dramatically due to the context switching overhead. The graphics card has much more cores than the CPU. Therefore, when the number of processes/threads exceeds 32, the speedup is stable.

Note that in the OpenMP version of the program, the speedup drops tremendously when the number of threads reaches 64. This might be related to the mutual exclusive locks to the global variable that indicates whether the program should terminate. When the number of threads exceeds the number of processor cores, the threads need to do time sharing. Therefore, active threads may need to wait for mutual exclusive locks obtained by inactive threads, which could be a huge waste of time.

Besides, the Jacobi algorithm has a higher performance than the Sor.

In terms of performance, the MPI 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 MPI 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 two different heat distribution 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.

Second, since the maximum processor cores of a single node of the cloud server is 32, it is not feasible to test larger number of cores for the MPI, pthread, and OpenMP versions of the program. 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 <chrono>
#include <hdist/hdist.hpp>
#include <mpi.h>

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

ImColor temp_to_color(double temp) {
    auto value = static_cast<uint8_t>(temp / 100.0 * 255.0);
    return {value, 0, 255 - value};
}

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);

    bool first = true;
    int finished = 0;
    int local_finished = finished;
    static hdist::State current_state, last_state;

    int num_iteration = 0;
    int max_iteration = 100;

    if (argc >= 2) {
        current_state.room_size = atoi(argv[1]);
        current_state.source_x = current_state.room_size / 2;
        current_state.source_y = current_state.room_size / 2;
    }
    if (argc >= 3) {
        max_iteration = atoi(argv[2]);
    }
    if (argc >= 4) {
```

```
        int algo = atoi(argv[3]);
        if (algo == 0) {
            current_state.algo = hdist::Algorithm::Jacobi;
        }
        else {
            current_state.algo = hdist::Algorithm::Sor;
        }
    }

    int chunk = (current_state.room_size + size - 1) / size;
    int capacity = chunk * size;

    double* global_buffer;
    auto grid = hdist::Grid{
        static_cast<size_t>(current_state.room_size),
        static_cast<size_t>(capacity),
        current_state.border_temp,
        current_state.source_temp,
        static_cast<size_t>(current_state.source_x),
        static_cast<size_t>(current_state.source_y)};

    if (0 == rank) {
        static std::chrono::high_resolution_clock::time_point begin,
end;
        static size_t duration = 0;
        static const char* algo_list[2] = { "jacobi", "sor" };
        graphic::GraphicContext context{"Assignment 4"};
        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 4", 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::DragInt("Room Size", &current_state.room_size, 10,
200, 1600, "%d");
```

```
            ImGui::DragFloat("Block Size", &current_state.block_size,
0.01, 0.1, 10, "%f");
            ImGui::DragFloat("Source Temp", &current_state.source_temp,
0.1, 0, 100, "%f");
            ImGui::DragFloat("Border Temp", &current_state.border_temp,
0.1, 0, 100, "%f");
            ImGui::DragInt("Source X", &current_state.source_x, 1, 1,
current_state.room_size - 2, "%d");
            ImGui::DragInt("Source Y", &current_state.source_y, 1, 1,
current_state.room_size - 2, "%d");
            ImGui::DragFloat("Tolerance", &current_state.tolerance,
0.01, 0.01, 1, "%f");
            ImGui::ListBox("Algorithm", reinterpret_cast<int
*>(&current_state.algo), algo_list, 2);

            if (current_state.algo == hdist::Algorithm::Sor) {
                ImGui::DragFloat("Sor Constant",
&current_state.sor_constant, 0.01, 0.0, 20.0, "%f");
            }

            // GUI paramater adjustment is disbled

            // if (current_state.room_size != last_state.room_size) {
            //     grid = hdist::Grid{
            //             static_cast<size_t>(current_state.room_size),
            //             current_state.border_temp,
            //             current_state.source_temp,
            //             static_cast<size_t>(current_state.source_x),
            //             static_cast<size_t>(current_state.source_y)};
            //     first = true;
            // }

            // if (current_state != last_state) {
            //     last_state = current_state;
            //     finished = false;
            // }

            if (first) {
                first = false;
                finished = 0;
            }

            if (!finished) {
                if (num_iteration < max_iteration) {
```

```
                    begin = std::chrono::high_resolution_clock::now();
                    if (size > 1) {
                        // synchronize the points on the boarder
                        // communicate with the succeeding process
                        // even rank (0): send before receive
                        MPI_Send(grid.get_current_buffer().data() +
(chunk - 1) * current_state.room_size,
                                    current_state.room_size, MPI_DOUBLE,
1, 0, MPI_COMM_WORLD);
                        MPI_Recv(grid.get_current_buffer().data() + chunk
* current_state.room_size,
                                    current_state.room_size, MPI_DOUBLE,
1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                        if (current_state.algo == hdist::Algorithm::Sor)
{
                            MPI_Send(grid.get_alt_buffer().data() +
(chunk - 1) * current_state.room_size,
                                        current_state.room_size,
MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);
                            MPI_Recv(grid.get_alt_buffer().data() + chunk
* current_state.room_size,
                                        current_state.room_size,
MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                        }
                    }
                    local_finished = (int)
hdist::calculate(current_state, grid, rank, chunk);
                    // synchronize the global status (finished or not)
                    MPI_Allreduce(&local_finished, &finished, 1,
MPI_INT, MPI_LAND, MPI_COMM_WORLD);
                    end = std::chrono::high_resolution_clock::now();
                    duration +=
duration_cast<std::chrono::nanoseconds>(end - begin).count();

                    // collect the results for drawing
                    // hence this does not count into the total time
                    global_buffer = grid.get_current_buffer().data();
                    MPI_Gather(grid.get_current_buffer().data() + rank *
chunk * current_state.room_size,
                                chunk * current_state.room_size,
MPI_DOUBLE, global_buffer,
                                chunk * current_state.room_size,
MPI_DOUBLE, 0, MPI_COMM_WORLD);
                    num_iteration ++;
```

```
                }
            }

            const ImVec2 p = ImGui::GetCursorScreenPos();
            float x = p.x + current_state.block_size, y = p.y +
current_state.block_size;

            for (size_t i = 0; i < current_state.room_size; ++i) {
                for (size_t j = 0; j < current_state.room_size; ++j) {
                    auto temp = grid[{i, j}];
                    auto color = temp_to_color(temp);
                    draw_list->AddRectFilled(ImVec2(x, y), ImVec2(x +
current_state.block_size, y + current_state.block_size), color);
                    y += current_state.block_size;
                }
                x += current_state.block_size;
                y = p.y + current_state.block_size;
            }

            if (finished) {
                ImGui::Text("stabilized in %lf ms", (double) duration /
1'000'000);
            }
            else if (num_iteration >= max_iteration) {
                ImGui::Text("%d iterations reached in %lf ms",
max_iteration, (double) duration / 1'000'000);
            }

            ImGui::End();
        });
    }
    else {
        while(num_iteration < max_iteration) {
            if (!finished) {
                // synchronize the points on the boarder
                // even rank: send before receive
                if (rank % 2 == 0) {
                    // communicate with the succeeding process
                    if (rank < size - 1) {
                        MPI_Send(grid.get_current_buffer().data() + (rank
* chunk + chunk - 1) * current_state.room_size,
                            current_state.room_size, MPI_DOUBLE,
rank + 1, 0, MPI_COMM_WORLD);
```

```
                        MPI_Recv(grid.get_current_buffer().data() + (rank
* chunk + chunk) * current_state.room_size,
                                    current_state.room_size, MPI_DOUBLE,
rank + 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                        if (current_state.algo == hdist::Algorithm::Sor)
{
                            MPI_Send(grid.get_alt_buffer().data() + (rank
* chunk + chunk - 1) * current_state.room_size,
                                        current_state.room_size,
MPI_DOUBLE, rank + 1, 0, MPI_COMM_WORLD);
                            MPI_Recv(grid.get_alt_buffer().data() + (rank
* chunk + chunk) * current_state.room_size,
                                        current_state.room_size,
MPI_DOUBLE, rank + 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                        }
                    }
                    // communicate with the preceding process
                    MPI_Send(grid.get_current_buffer().data() + (rank *
chunk) * current_state.room_size,
                                current_state.room_size, MPI_DOUBLE, rank
- 1, 0, MPI_COMM_WORLD);
                    MPI_Recv(grid.get_current_buffer().data() + (rank *
chunk - 1) * current_state.room_size,
                                current_state.room_size, MPI_DOUBLE, rank
- 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                    if (current_state.algo == hdist::Algorithm::Sor) {
                        MPI_Send(grid.get_alt_buffer().data() + (rank *
chunk) * current_state.room_size,
                                current_state.room_size, MPI_DOUBLE, rank
- 1, 0, MPI_COMM_WORLD);
                        MPI_Recv(grid.get_alt_buffer().data() + (rank *
chunk - 1) * current_state.room_size,
                                current_state.room_size, MPI_DOUBLE, rank
- 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                    }
                }
                // odd rank: receive before send
                else {
                    // communicate with the preceding process
                    MPI_Recv(grid.get_current_buffer().data() + (rank *
chunk - 1) * current_state.room_size,
                                current_state.room_size, MPI_DOUBLE, rank
- 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

```
                    MPI_Send(grid.get_current_buffer().data() + (rank *
chunk) * current_state.room_size,
                                current_state.room_size, MPI_DOUBLE, rank
- 1, 0, MPI_COMM_WORLD);
                    if (current_state.algo == hdist::Algorithm::Sor) {
                        MPI_Recv(grid.get_alt_buffer().data() + (rank *
chunk - 1) * current_state.room_size,
                                current_state.room_size, MPI_DOUBLE, rank
- 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                        MPI_Send(grid.get_alt_buffer().data() + (rank *
chunk) * current_state.room_size,
                                current_state.room_size, MPI_DOUBLE, rank
- 1, 0, MPI_COMM_WORLD);
                    }
                    // communicate with the succeeding process
                    if (rank < size - 1) {
                        MPI_Send(grid.get_current_buffer().data() + (rank
* chunk + chunk - 1) * current_state.room_size,
                                    current_state.room_size, MPI_DOUBLE,
rank + 1, 0, MPI_COMM_WORLD);
                        MPI_Recv(grid.get_current_buffer().data() + (rank
* chunk + chunk) * current_state.room_size,
                                    current_state.room_size, MPI_DOUBLE,
rank + 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                        if (current_state.algo == hdist::Algorithm::Sor)
{
                            MPI_Send(grid.get_alt_buffer().data() + (rank
* chunk + chunk - 1) * current_state.room_size,
                                    current_state.room_size, MPI_DOUBLE,
rank + 1, 0, MPI_COMM_WORLD);
                            MPI_Recv(grid.get_alt_buffer().data() + (rank
* chunk + chunk) * current_state.room_size,
                                    current_state.room_size, MPI_DOUBLE,
rank + 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                        }
                    }
                }
                local_finished = (int) hdist::calculate(current_state,
grid, rank, chunk);
                // synchronize the global status (finished or not)
                MPI_Allreduce(&local_finished, &finished, 1, MPI_INT,
MPI_LAND, MPI_COMM_WORLD);
                MPI_Gather(grid.get_current_buffer().data() + rank *
chunk * current_state.room_size,
```

```
                    chunk * current_state.room_size, MPI_DOUBLE,
global_buffer,
                    chunk * current_state.room_size, MPI_DOUBLE, 0,
MPI_COMM_WORLD);
            }
            num_iteration ++;
        }
    }
}
```

2. Pthread Version

```
#include <graphic/graphic.hpp>
#include <imgui_impl_sdl.h>
#include <cstring>
#include <chrono>
#include <hdist/hdist.hpp>

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

ImColor temp_to_color(double temp) {
    auto value = static_cast<uint8_t>(temp / 100.0 * 255.0);
    return {value, 0, 255 - value};
}

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_barrier_t barrier;

hdist::Grid* grid;

bool first = true;
int finished = 0;
int local_finished = finished;
hdist::State current_state, last_state;
int max_iteration = 100;
int chunk = 0;

void* gui_loop(void* t) {

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

```
    int num_iteration = 0;

    if (0 == tid) {
        static std::chrono::high_resolution_clock::time_point begin,
end;
        static size_t duration = 0;
        static const char* algo_list[2] = { "jacobi", "sor" };
        graphic::GraphicContext context{"Assignment 4"};
        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 4", 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::DragInt("Room Size", &current_state.room_size, 10,
200, 1600, "%d");
            ImGui::DragFloat("Block Size", &current_state.block_size,
0.01, 0.1, 10, "%f");
            ImGui::DragFloat("Source Temp", &current_state.source_temp,
0.1, 0, 100, "%f");
            ImGui::DragFloat("Border Temp", &current_state.border_temp,
0.1, 0, 100, "%f");
            ImGui::DragInt("Source X", &current_state.source_x, 1, 1,
current_state.room_size - 2, "%d");
            ImGui::DragInt("Source Y", &current_state.source_y, 1, 1,
current_state.room_size - 2, "%d");
            ImGui::DragFloat("Tolerance", &current_state.tolerance,
0.01, 0.01, 1, "%f");
            ImGui::ListBox("Algorithm", reinterpret_cast<int
*>(&current_state.algo), algo_list, 2);

            if (current_state.algo == hdist::Algorithm::Sor) {
                ImGui::DragFloat("Sor Constant",
&current_state.sor_constant, 0.01, 0.0, 20.0, "%f");
            }
```

```
            // GUI paramater adjustment is disbled

            // if (current_state.room_size != last_state.room_size) {
            //     grid = hdist::Grid{
            //             static_cast<size_t>(current_state.room_size),
            //             current_state.border_temp,
            //             current_state.source_temp,
            //             static_cast<size_t>(current_state.source_x),
            //             static_cast<size_t>(current_state.source_y)};
            //     first = true;
            // }

            // if (current_state != last_state) {
            //     last_state = current_state;
            //     finished = false;
            // }

            if (first) {
                first = false;
                finished = 0;
            }

            if (!finished) {
                if (num_iteration < max_iteration) {
                    begin = std::chrono::high_resolution_clock::now();
                    if (current_state.algo == hdist::Algorithm::Jacobi)
{
                        local_finished = (int)
hdist::calculate_jacobi(current_state, *grid, tid, chunk);
                    }
                    else if (current_state.algo ==
hdist::Algorithm::Sor) {
                        local_finished = (int)
hdist::calculate_sor(current_state, *grid, tid, chunk, 0);
                        // sync threads and switch buffer
                        pthread_barrier_wait(&barrier);
                        grid->switch_buffer();
                        // acknowledge other threads that the buffer has
been switched
                        pthread_barrier_wait(&barrier);
                        local_finished = (int)
hdist::calculate_sor(current_state, *grid, tid, chunk, 1);
                    }
                    pthread_mutex_lock(&mutex);
```

```
                    // update the global status (finished or not)
                    finished &= local_finished;
                    pthread_mutex_unlock(&mutex);
                    // sync threads and switch buffer
                    pthread_barrier_wait(&barrier);
                    grid->switch_buffer();
                    // acknowledge other threads that the buffer has
been switched
                    pthread_barrier_wait(&barrier);

                    // hence this does not count into the total time
                    end = std::chrono::high_resolution_clock::now();
                    duration +=
duration_cast<std::chrono::nanoseconds>(end - begin).count();
                    num_iteration ++;
                }
            }

            const ImVec2 p = ImGui::GetCursorScreenPos();
            float x = p.x + current_state.block_size, y = p.y +
current_state.block_size;

            for (size_t i = 0; i < current_state.room_size; ++i) {
                for (size_t j = 0; j < current_state.room_size; ++j) {
                    auto temp = (*grid)[{i, j}];
                    auto color = temp_to_color(temp);
                    draw_list->AddRectFilled(ImVec2(x, y), ImVec2(x +
current_state.block_size, y + current_state.block_size), color);
                    y += current_state.block_size;
                }
                x += current_state.block_size;
                y = p.y + current_state.block_size;
            }

            if (finished) {
                ImGui::Text("stabilized in %lf ms", (double) duration /
1'000'000);
            }
            else if (num_iteration >= max_iteration) {
                ImGui::Text("%d iterations reached in %lf ms",
max_iteration, (double) duration / 1'000'000);
            }

            ImGui::End();
```

```
        });
    }
    else {
        while (num_iteration < max_iteration) {
            if (!finished) {
                if (current_state.algo == hdist::Algorithm::Jacobi) {
                    local_finished = (int)
hdist::calculate_jacobi(current_state, *grid, tid, chunk);
                }
                else if (current_state.algo == hdist::Algorithm::Sor) {
                    local_finished = (int)
hdist::calculate_sor(current_state, *grid, tid, chunk, 0);
                    // sync threads for switching buffer
                    pthread_barrier_wait(&barrier);
                    // wait thread 0 to switch buffer
                    pthread_barrier_wait(&barrier);
                    local_finished = (int)
hdist::calculate_sor(current_state, *grid, tid, chunk, 1);
                }
                pthread_mutex_lock(&mutex);
                // update the global status (finished or not)
                finished &= local_finished;
                pthread_mutex_unlock(&mutex);
                // sync threads for switching buffer
                pthread_barrier_wait(&barrier);
                // wait thread 0 to switch buffer
                pthread_barrier_wait(&barrier);
            }
            num_iteration ++;
        }
    }
    return nullptr;
}

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

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

    if (argc >= 3) {
        current_state.room_size = atoi(argv[2]);
        current_state.source_x = current_state.room_size / 2;
```

```
        current_state.source_y = current_state.room_size / 2;
    }
    if (argc >= 4) {
        max_iteration = atoi(argv[3]);
    }
    if (argc >= 5) {
        int algo = atoi(argv[4]);
        if (algo == 0) {
            current_state.algo = hdist::Algorithm::Jacobi;
        }
        else {
            current_state.algo = hdist::Algorithm::Sor;
        }
    }

    chunk = (current_state.room_size + num_threads - 1) / num_threads;
    int capacity = chunk * num_threads;

    grid = new hdist::Grid{
        static_cast<size_t>(current_state.room_size),
        static_cast<size_t>(capacity),
        current_state.border_temp,
        current_state.source_temp,
        static_cast<size_t>(current_state.source_x),
        static_cast<size_t>(current_state.source_y)};

    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 <chrono>
#include <hdist/hdist.hpp>
#include <omp.h>

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

ImColor temp_to_color(double temp) {
    auto value = static_cast<uint8_t>(temp / 100.0 * 255.0);
    return {value, 0, 255 - value};
}

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

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

    hdist::State current_state, last_state;
    bool first = true;
    int finished = 0;
    int local_finished = finished;
    int max_iteration = 100;
    int chunk = 0;

    if (argc >= 3) {
        current_state.room_size = atoi(argv[2]);
        current_state.source_x = current_state.room_size / 2;
        current_state.source_y = current_state.room_size / 2;
    }
    if (argc >= 4) {
        max_iteration = atoi(argv[3]);
    }
    if (argc >= 5) {
        int algo = atoi(argv[4]);
        if (algo == 0) {
            current_state.algo = hdist::Algorithm::Jacobi;
        }
```

```
        else {
            current_state.algo = hdist::Algorithm::Sor;
        }
    }

    chunk = (current_state.room_size + n_threads - 1) / n_threads;
    int capacity = chunk * n_threads;

    hdist::Grid grid(
        static_cast<size_t>(current_state.room_size),
        static_cast<size_t>(capacity),
        current_state.border_temp,
        current_state.source_temp,
        static_cast<size_t>(current_state.source_x),
        static_cast<size_t>(current_state.source_y));

    omp_lock_t lock;
    omp_init_lock(&lock);

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

        if (0 == tid) {
            static std::chrono::high_resolution_clock::time_point begin,
end;
            static size_t duration = 0;
            static const char* algo_list[2] = { "jacobi", "sor" };
            graphic::GraphicContext context{"Assignment 4"};
            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 4", 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::DragInt("Room Size", &current_state.room_size,
10, 200, 1600, "%d");
                ImGui::DragFloat("Block Size",
&current_state.block_size, 0.01, 0.1, 10, "%f");
                ImGui::DragFloat("Source Temp",
&current_state.source_temp, 0.1, 0, 100, "%f");
                ImGui::DragFloat("Border Temp",
&current_state.border_temp, 0.1, 0, 100, "%f");
                ImGui::DragInt("Source X", &current_state.source_x, 1,
1, current_state.room_size - 2, "%d");
                ImGui::DragInt("Source Y", &current_state.source_y, 1,
1, current_state.room_size - 2, "%d");
                ImGui::DragFloat("Tolerance", &current_state.tolerance,
0.01, 0.01, 1, "%f");
                ImGui::ListBox("Algorithm", reinterpret_cast<int
*>(&current_state.algo), algo_list, 2);

                if (current_state.algo == hdist::Algorithm::Sor) {
                    ImGui::DragFloat("Sor Constant",
&current_state.sor_constant, 0.01, 0.0, 20.0, "%f");
                }

                // GUI paramater adjustment is disbled

                // if (current_state.room_size != last_state.room_size)
{
                //     grid = hdist::Grid{
                //             static_cast<size_t>(current_state.room_siz
e),
                //             current_state.border_temp,
                //             current_state.source_temp,
                //             static_cast<size_t>(current_state.source_x
),
                //             static_cast<size_t>(current_state.source_y
)};
                //     first = true;
                // }

                // if (current_state != last_state) {
                //     last_state = current_state;
                //     finished = false;
                // }

                if (first) {
```

```
                    first = false;
                    finished = 0;
                }

                if (!finished) {
                    if (num_iteration < max_iteration) {
                        begin =
std::chrono::high_resolution_clock::now();
                        if (current_state.algo ==
hdist::Algorithm::Jacobi) {
                            local_finished = (int)
hdist::calculate_jacobi(current_state, grid, tid, chunk);
                        }
                        else if (current_state.algo ==
hdist::Algorithm::Sor) {
                            local_finished = (int)
hdist::calculate_sor(current_state, grid, tid, chunk, 0);
                            // sync threads and switch buffer
                            #pragma omp barrier
                            grid.switch_buffer();
                            // acknowledge other threads that the buffer
has been switched
                            #pragma omp barrier
                            local_finished = (int)
hdist::calculate_sor(current_state, grid, tid, chunk, 1);
                        }
                        omp_set_lock(&lock);
                        // update the global status (finished or not)
                        finished &= local_finished;
                        omp_unset_lock(&lock);
                        // sync threads and switch buffer
                        #pragma omp barrier
                        grid.switch_buffer();
                        // acknowledge other threads that the buffer has
been switched
                        #pragma omp barrier

                        // hence this does not count into the total time
                        end = std::chrono::high_resolution_clock::now();
                        duration +=
duration_cast<std::chrono::nanoseconds>(end - begin).count();
                        num_iteration ++;
                    }
                }
```

```
                const ImVec2 p = ImGui::GetCursorScreenPos();
                float x = p.x + current_state.block_size, y = p.y +
current_state.block_size;

                for (size_t i = 0; i < current_state.room_size; ++i) {
                    for (size_t j = 0; j < current_state.room_size; ++j)
{
                        auto temp = grid[{i, j}];
                        auto color = temp_to_color(temp);
                        draw_list->AddRectFilled(ImVec2(x, y), ImVec2(x +
current_state.block_size, y + current_state.block_size), color);
                        y += current_state.block_size;
                    }
                    x += current_state.block_size;
                    y = p.y + current_state.block_size;
                }

                if (finished) {
                    ImGui::Text("stabilized in %lf ms", (double)
duration / 1'000'000);
                }
                else if (num_iteration >= max_iteration) {
                    ImGui::Text("%d iterations reached in %lf ms",
max_iteration, (double) duration / 1'000'000);
                }

                ImGui::End();
            });
        }
        else {
            while (num_iteration < max_iteration) {
                if (!finished) {
                    if (current_state.algo == hdist::Algorithm::Jacobi)
{
                        local_finished = (int)
hdist::calculate_jacobi(current_state, grid, tid, chunk);
                    }
                    else if (current_state.algo ==
hdist::Algorithm::Sor) {
                        local_finished = (int)
hdist::calculate_sor(current_state, grid, tid, chunk, 0);
                        // sync threads for switching buffer
                        #pragma omp barrier
```

```
                        // wait thread 0 to switch buffer
                        #pragma omp barrier
                        local_finished = (int)
hdist::calculate_sor(current_state, grid, tid, chunk, 1);
                    }
                    omp_set_lock(&lock);
                    // update the global status (finished or not)
                    finished &= local_finished;
                    omp_unset_lock(&lock);
                    // sync threads for switching buffer
                    #pragma omp barrier
                    // wait thread 0 to switch buffer
                    #pragma omp barrier
                }
                num_iteration ++;
            }
        }
    }

    return 0;
}
```

4. CUDA Version

```
#include <graphic/graphic.hpp>
#include <imgui_impl_sdl.h>
#include <cstring>
#include <chrono>
#include <hdist/hdist.hpp>
#include <cuda.h>
#include <cuda_runtime.h>
#include <device_launch_parameters.h>
#include <inttypes.h>

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

ImColor temp_to_color(double temp) {
    auto value = static_cast<uint8_t>(temp / 100.0 * 255.0);
    return {value, 0, 255 - value};
}
```

```
__global__ void calculate(hdist::Grid* grid,
                          hdist::State* current_state,
                          int* chunk,
                          bool* finished,
                          int* lock) {
    int tid = threadIdx.x;
    bool local_finished = false;
    if (current_state->algo == hdist::Algorithm::Jacobi) {
        local_finished = hdist::calculate_jacobi(current_state, grid,
tid, *chunk);
    }
    else if (current_state->algo == hdist::Algorithm::Sor) {
        local_finished = hdist::calculate_sor(current_state, grid, tid,
*chunk, 0);
        // sync threads and switch buffer
        __syncthreads();
        if (tid == 0) {
            grid->switch_buffer();
        }
        // acknowledge other threads that the buffer has been switched
        __syncthreads();
        local_finished = hdist::calculate_sor(current_state, grid, tid,
*chunk, 1);
    }
    while (*lock != 0) {};
    *lock = 1;
    // update the global status (finished or not)
    *finished &= local_finished;
    *lock = 0;
    // sync threads and switch buffer
    __syncthreads();
    if (tid == 0) {
        grid->switch_buffer();
    }
}

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

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

    bool first = true;
```

```
    bool finished = false;
    hdist::State current_state; //, last_state;
    int max_iteration = 100;
    int chunk = 0;
    int num_iteration = 0;

    if (argc >= 3) {
        current_state.room_size = atoi(argv[2]);
        current_state.source_x = current_state.room_size / 2;
        current_state.source_y = current_state.room_size / 2;
    }
    if (argc >= 4) {
        max_iteration = atoi(argv[3]);
    }
    if (argc >= 5) {
        int algo = atoi(argv[4]);
        if (algo == 0) {
            current_state.algo = hdist::Algorithm::Jacobi;
        }
        else {
            current_state.algo = hdist::Algorithm::Sor;
        }
    }

    chunk = (current_state.room_size + num_threads - 1) / num_threads;
    int capacity = chunk * num_threads;

    auto grid = hdist::Grid{
        static_cast<size_t>(current_state.room_size),
        static_cast<size_t>(capacity),
        current_state.border_temp,
        current_state.source_temp,
        static_cast<size_t>(current_state.source_x),
        static_cast<size_t>(current_state.source_y)};

    cudaError_t cudaStatus;
    hdist::Grid* cuda_grid;
    hdist::State* cuda_current_state;
    int* cuda_chunk;
    bool* cuda_finished;
    int* cuda_lock;
    double* cuda_data0;
    double* cuda_data1;
```

```
    size_t cuda_current_buffer;
    int array_size = current_state.room_size * capacity *
sizeof(double);

    cudaMallocManaged((void**) &cuda_grid, sizeof(grid));
    cudaMemcpy(cuda_grid, &grid, sizeof(grid), cudaMemcpyHostToDevice);
    cudaMallocManaged((void**) &cuda_current_state,
sizeof(current_state));
    cudaMemcpy(cuda_current_state, &current_state,
sizeof(current_state), cudaMemcpyHostToDevice);
    cudaMallocManaged((void**) &cuda_chunk, sizeof(int));
    cudaMemcpy(cuda_chunk, &chunk, sizeof(int), cudaMemcpyHostToDevice);
    cudaMallocManaged((void**) &cuda_finished, sizeof(bool));
    cudaMemcpy(cuda_finished, &finished, sizeof(bool),
cudaMemcpyHostToDevice);
    cudaMallocManaged((void**) &cuda_lock, sizeof(int));
    cudaMemset(cuda_lock, 0, sizeof(int));
    cudaMallocManaged((void**) &cuda_data0, array_size);
    cudaMallocManaged((void**) &cuda_data1, array_size);
    cuda_grid->data0 = cuda_data0;
    cuda_grid->data1 = cuda_data1;
    cudaMemcpy(cuda_grid->data0, grid.data0, array_size,
cudaMemcpyHostToDevice);
    cudaMemcpy(cuda_grid->data1, grid.data1, array_size,
cudaMemcpyHostToDevice);

    static std::chrono::high_resolution_clock::time_point begin, end;
    static size_t duration = 0;
    static const char* algo_list[2] = { "jacobi", "sor" };
    graphic::GraphicContext context{"Assignment 4"};
    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 4", 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::DragInt("Room Size", &current_state.room_size, 10, 200,
1600, "%d");
        ImGui::DragFloat("Block Size", &current_state.block_size, 0.01,
0.1, 10, "%f");
        ImGui::DragFloat("Source Temp", &current_state.source_temp, 0.1,
0, 100, "%f");
        ImGui::DragFloat("Border Temp", &current_state.border_temp, 0.1,
0, 100, "%f");
        ImGui::DragInt("Source X", &current_state.source_x, 1, 1,
current_state.room_size - 2, "%d");
        ImGui::DragInt("Source Y", &current_state.source_y, 1, 1,
current_state.room_size - 2, "%d");
        ImGui::DragFloat("Tolerance", &current_state.tolerance, 0.01,
0.01, 1, "%f");
        ImGui::ListBox("Algorithm", reinterpret_cast<int
*>(&current_state.algo), algo_list, 2);

        if (current_state.algo == hdist::Algorithm::Sor) {
            ImGui::DragFloat("Sor Constant",
&current_state.sor_constant, 0.01, 0.0, 20.0, "%f");
        }

        // GUI paramater adjustment is disbled

        // if (current_state.room_size != last_state.room_size) {
        //     grid = hdist::Grid{
        //             static_cast<size_t>(current_state.room_size),
        //             current_state.border_temp,
        //             current_state.source_temp,
        //             static_cast<size_t>(current_state.source_x),
        //             static_cast<size_t>(current_state.source_y)};
        //     first = true;
        // }

        // if (current_state != last_state) {
        //     last_state = current_state;
        //     finished = false;
        // }

        if (first) {
            first = false;
            finished = 0;
        }
```

```
        if (!finished) {
            if (num_iteration < max_iteration) {
                begin = std::chrono::high_resolution_clock::now();

                // Launch GPU kernel
                calculate<<<1, num_threads>>>(cuda_grid,
                                              cuda_current_state,
                                              cuda_chunk,
                                              cuda_finished,
                                              cuda_lock);
                // Wait GPU to complete calculation
                cudaDeviceSynchronize();
                cudaStatus = cudaGetLastError();
                if (cudaStatus != cudaSuccess) {
                    fprintf(stderr, "mykernel launch failed: %s\n",
                            cudaGetErrorString(cudaStatus));
                    return 0;
                }
                // Get the finished status
                cudaMemcpy(&finished, cuda_finished, sizeof(bool),
cudaMemcpyDeviceToHost);
                end = std::chrono::high_resolution_clock::now();
                duration +=
std::chrono::duration_cast<std::chrono::nanoseconds>(end -
begin).count();
                num_iteration ++;
            }
        }

        const ImVec2 p = ImGui::GetCursorScreenPos();
        float x = p.x + current_state.block_size, y = p.y +
current_state.block_size;


        // Receive data from device
        cudaMemcpy(&cuda_current_buffer, &cuda_grid->current_buffer,
                   sizeof(size_t), cudaMemcpyDeviceToHost);
        if (cuda_current_buffer == 0)
        {
            cudaMemcpy(grid.get_current_buffer(), cuda_grid->data0,
                       array_size, cudaMemcpyDeviceToHost);
        }
        else {
            cudaMemcpy(grid.get_current_buffer(), cuda_grid->data1,
```

```
                        array_size, cudaMemcpyDeviceToHost);
        }

        for (size_t i = 0; i < current_state.room_size; ++i) {
            for (size_t j = 0; j < current_state.room_size; ++j) {
                auto temp = grid[{i, j}];
                auto color = temp_to_color(temp);
                draw_list->AddRectFilled(ImVec2(x, y), ImVec2(x +
current_state.block_size, y + current_state.block_size), color);
                y += current_state.block_size;
            }
            x += current_state.block_size;
            y = p.y + current_state.block_size;
        }

        if (finished) {
            ImGui::Text("stabilized in %lf ms", (double) duration /
1'000'000);
        }
        else if (num_iteration >= max_iteration) {
            ImGui::Text("%d iterations reached in %lf ms",
max_iteration, (double) duration / 1'000'000);
        }

        ImGui::End();
    });

    return 0;
}
```