Assignment Report

CSC 4005 Mandelbrot Set Computation

Wei Wu (吴畏)

118010335

October 31, 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 Mandelbrot Set calculator and visualizer using MPI and pthread. A Mandelbrot Set is defined as follows:

Set of points in a complex plane that are quasi-stable (will increase and decrease, but not exceed some limit) when computed by iterating the function: $z_{k+1} = z_k^2 + c$

when z_{k+1} is the (k+1)th iteration of the complex number $z = a + bi$ and c is a complex number giving the position of the point in the complex plan. (For example, in a image of height H and width W,

$$c = \frac{x - \text{height}/2}{\text{height}/4} + \frac{y - \text{width}/2}{\text{width}/4} \times i$$

You can also scale the to obtain a different output).

The initial value for z_0 is zero. The iterations continued until the magnitude of z_k is greater than a threshold or the maximum number of iterations have been achieved. For $z_k = a + bi$. The magnitude of z_k is defined below: $z_k = \sqrt{a^2 + b^2}$

Computing the complex function $z_{k+1} = z_k^2 + c$ is simplified by recognizing that:

$$z^2 = a^2 + 2abi + bi^2 = a^2 - b^2 + 2abi$$

Therefore, real part is the $a^2 - b^2$ while the imaginary part is $2abi$.

The next iteration values can be produced by computing:

$$z_{\text{real}} = z_{\text{real}}^2 - z_{\text{imag}}^2 + c_{\text{real}} \quad z_{\text{imag}} = 2z_{\text{real}}\, z_{\text{imag}} + c_{\text{imag}}$$

II. Design Approaches

1. Architecture

a. MPI

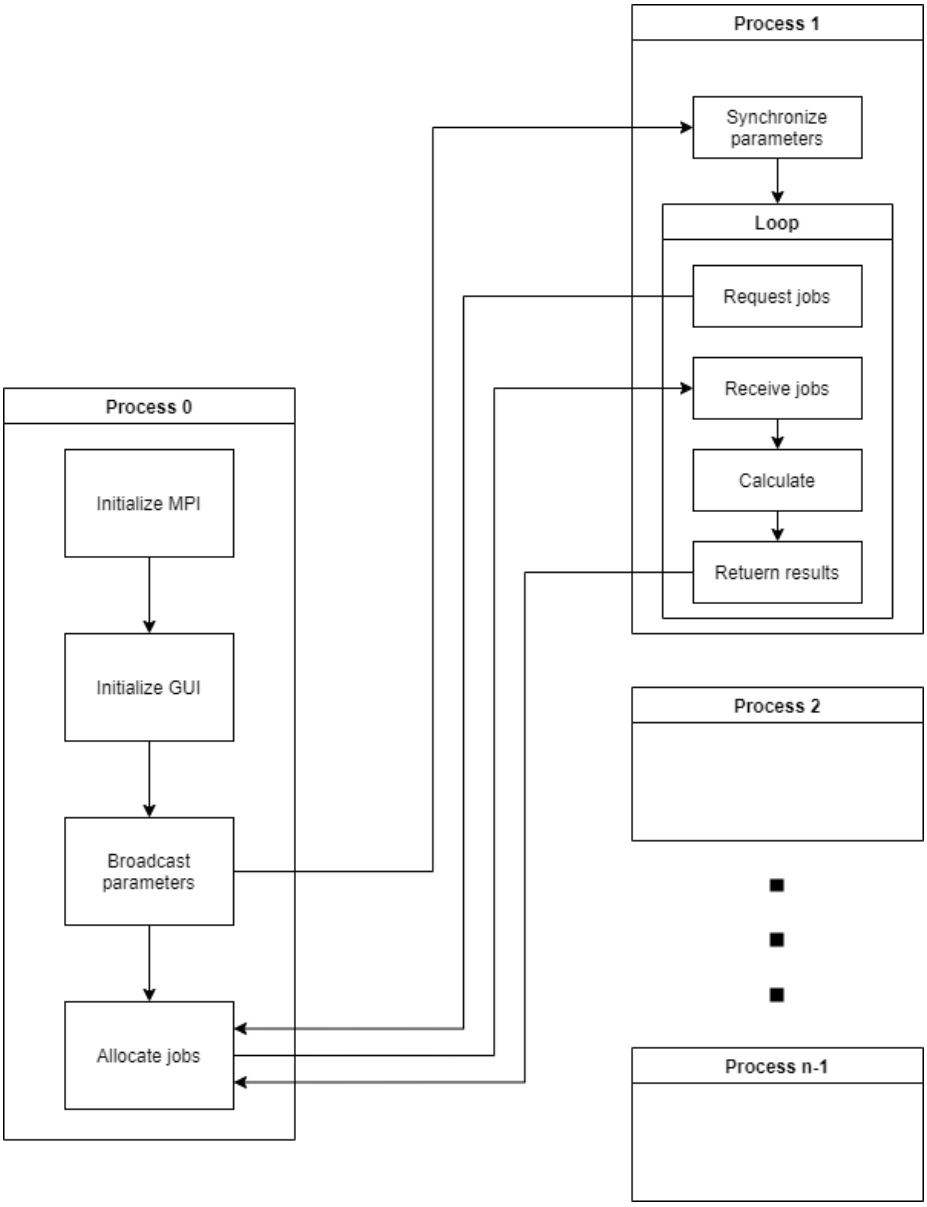

Figure 1. Architecture of the MPI version of the program

The MPI version of the program adopts a master-slave organization. When only one process is launched, the program executes the sequential algorithm directly. When more than one

processes are launched, the root process acts as the master, while all the other processes act as slaves. In each drawing iteration, the master broadcasts the parameters, including size, k_value, center_x, and center_y, to all the processes. Then, whenever a slave is ready, it sends a message to the master for acknowledgement. When the whole calculation is not finished, the master allocates a chunk of points (a certain number of rows) to the slave from which the acknowledgement is received. Next, the slave who gets the allocation performs calculation. When the calculation is done, the slave notifies the master and sends the results to it. If the whole calculation is still not finished, the master allocates another chunk to the slave. The loop continues until all the calculation is done.

b. pthread

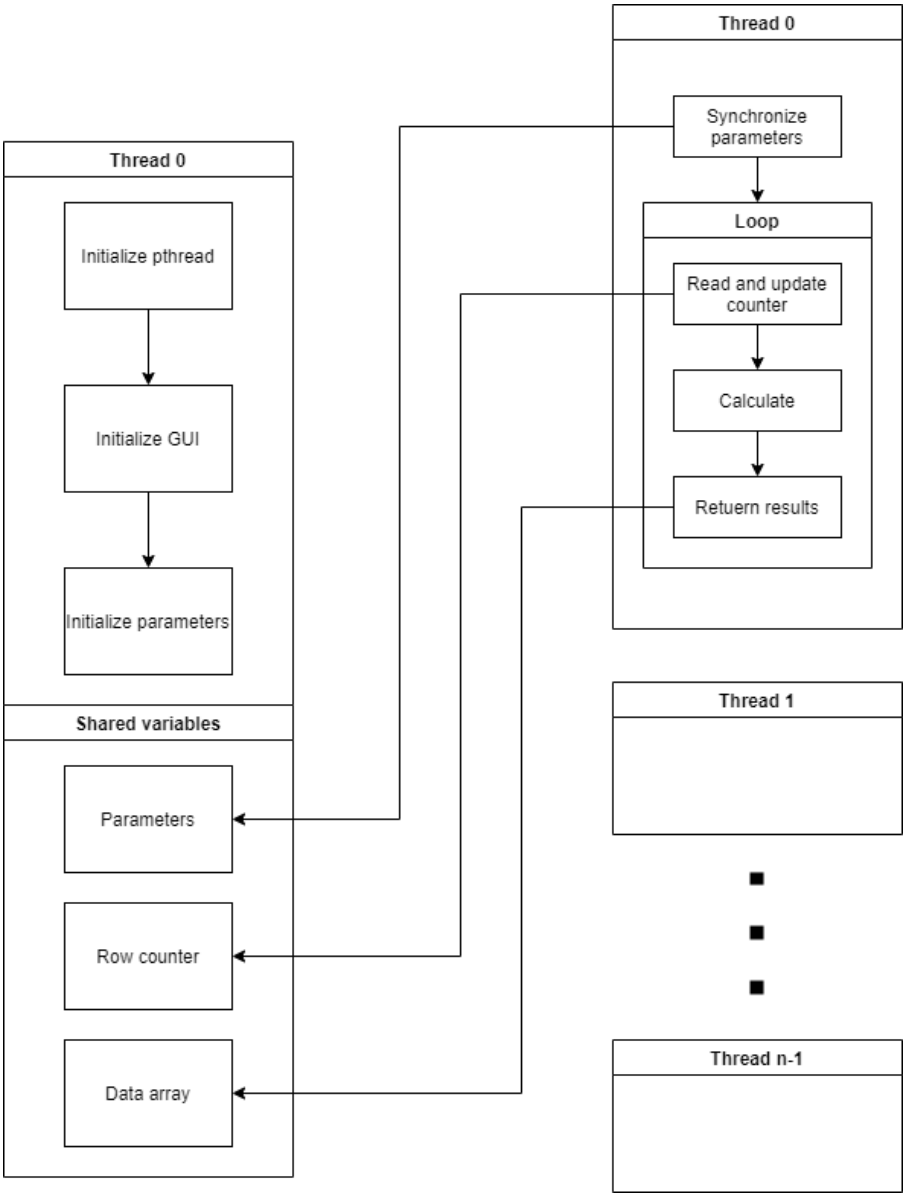

Figure 2. Architecture of the pthread version of the program

The pthread version of the program adopts the shared memory approach. Therefore, the root thread only initializes the pthread framework, GUI, and shared variables. Unlike the MPI version, the root thread does not serve as a master thread to coordinate other threads. Instead, in each iteration of drawing, the root thread

updates the parameters, row counter, and data array in the shared memory. Then, all the threads (including the root) read the shared parameters for update. Next, each thread reads the row counter to get its batch of points for processing and update the row counter. After the calculation, the thread writes the results back to the shared data array. The thread that finishes its batch checks the row counter to see if there is any other batch. If yes, it fetches the row number again and starts calculation. Otherwise, it suspends until the next iteration of drawing.

2. Data Structures

a. MPI

(i) int[4] int_params_buffer

Used to contain parameters of integer type for broadcasting.

int_params_buffer[0]: k_value

int_params_buffer[1]: scale

int_params_buffer[2]: size

int_params_buffer[3]: chunk_size

(ii) double[2] double_params_buffer

Used to contain parameters of double type for broadcasting.

double_params_buffer[0]: center_x

double_params_buffer[1]: center_y

(iii) int* global_buffer

Used to contain the complete results in the root process.

(iv) int* local_buffer

Used to contain the results of a batch in the slave processes.

b. pthread

(i) global_row_number

Used to indicate the next row number for processing in the shared memory.

(ii) row_number

Used to indicate the current row number to process in the thread-local memory.

3. Algorithm

In Mandelbrot Set, different points require different numbers of iterations of calculation. There are more quasi-stable points around the center of the square. That is, the points around the center of the square need more iterations. If we split the square into pieces, with one piece for each process/thread, the

workload will be extremely unbalanced. The processes/threads are expected to have huge differences in processing time. Due to the Cask Effect, the total execution time depends on the longest execution time among all processes/threads. Therefore, we can split the square into much more pieces (batches) than the number of processes/threads. The process/thread gets allocated and calculates one batch at a time. When a process/thread finishes its allocated batch, it fetches another batch from the coordinator or global counter for processing. In this way, the batches can be distributed more evenly among the processes, leading to a higher speedup.

III. Build And Run

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 MPI version program, use the following commands on the server:

```
mpirun csc4005_imgui (size) (k_value) (chunk_size)
```

where size is the size of the square, k_value is the value of parameter k, chunk_size is the number of columns in each chunk.

Examples:

a. mpirun csc4005_imgui
b. mpirun csc4005_imgui 800 100
c. mpirun csc4005_imgui 800 100 5

Note: When there is only one process, the program behaves the same as the sequential algorithm. When there are more than one processes, one process will work as a master process while others work as the slave processes.

```
bash-4.2$ mpirun csc4005_imgui 400 100 10
[WARN] cannot get screen dpi, auto-scaling disabled
libunwind: __unw_add_dynamic_fde: bad fde: FDE is really a CIE
libunwind: __unw_add_dynamic_fde: bad fde: FDE is really a CIE
libunwind: __unw_add_dynamic_fde: bad fde: FDE is really a CIE
7600 pixels in last 500777944 nanoseconds
speed: 15176.4 pixels per second
8000 pixels in last 525286458 nanoseconds
speed: 15229.8 pixels per second
```

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

srun -n1 csc4005_imgui num_threads (size) (k_value) (chunk_size)

where num_threads is the number of threads to run, size is the size of the square, k_value is the value of parameter k, chunk_size is the number of columns in each chunk.

Examples:

a. srun -n1 csc4005 4

b. srun -n1 csc4005 4 800 100

c. srun -n1 csc4005 4 800 100 5

```
bash-4.2$ srun -n1 csc4005_imgui 32 3200 100 10
[WARN] cannot get screen dpi, auto-scaling disabled
libunwind: __unw_add_dynamic_fde: bad fde: FDE is really a CIE
libunwind: __unw_add_dynamic_fde: bad fde: FDE is really a CIE
libunwind: __unw_add_dynamic_fde: bad fde: FDE is really a CIE
25600 pixels in last 507445366 nanoseconds
speed: 50448.8 pixels per second
25600 pixels in last 505181868 nanoseconds
speed: 50674.8 pixels per second
```

IV. Performance Analysis

1. Test Results

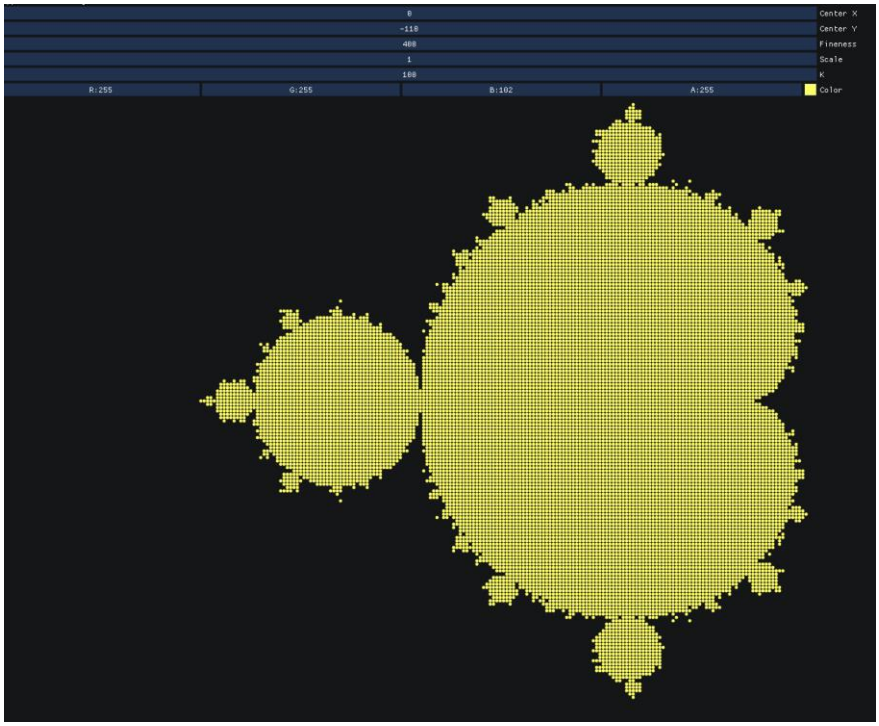

Figure 3. GUI Output for size=400, k=100.

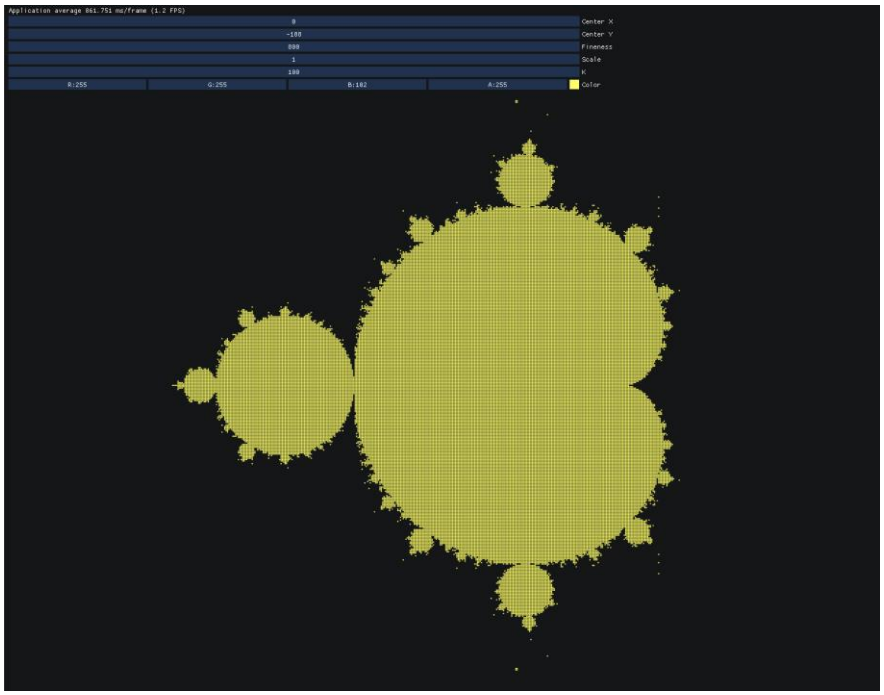

Figure 4. GUI Output for size=800, k=100.

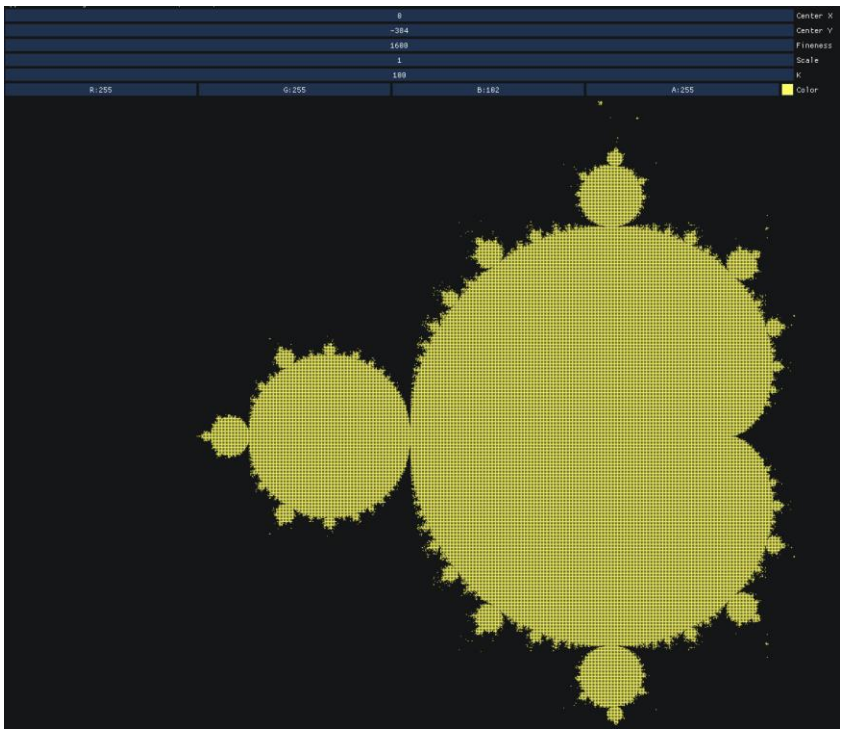

Figure 5. GUI Output for size=1600, k=100.

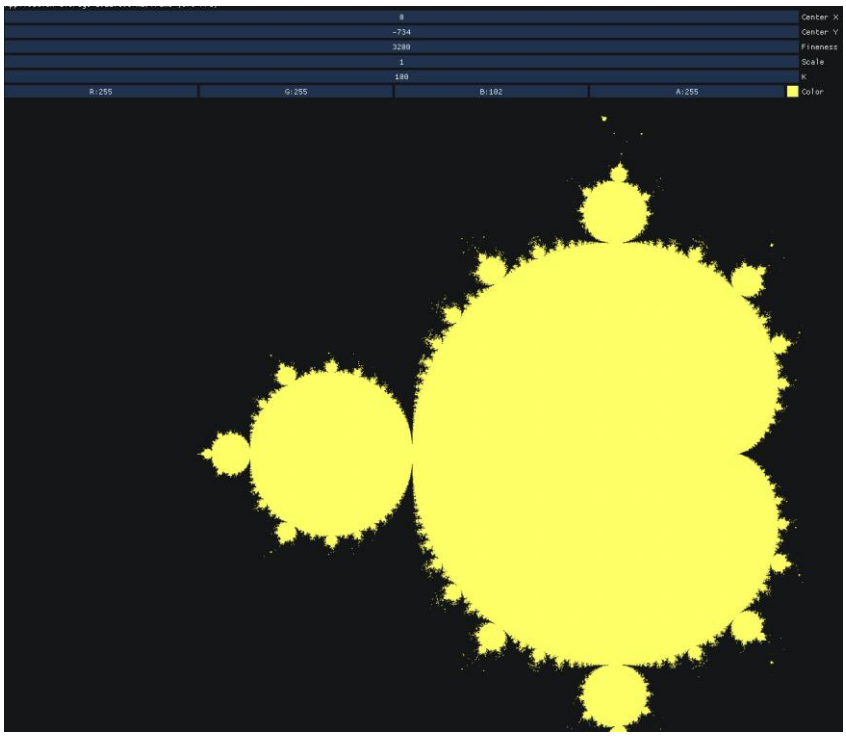

Figure 6. GUI Output for size=3200, k=100.

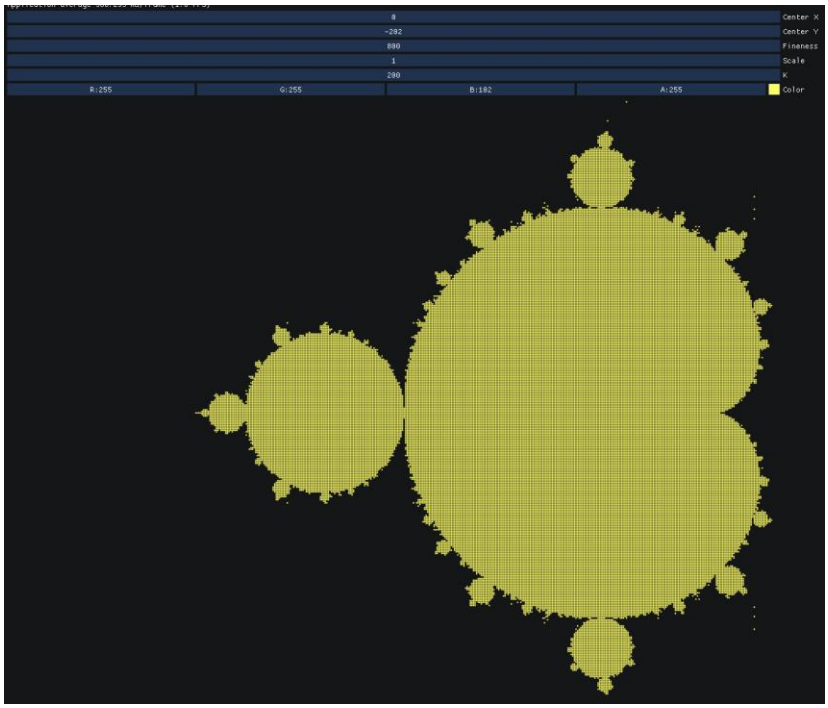

Figure 7. GUI Output for size=800, k=200.

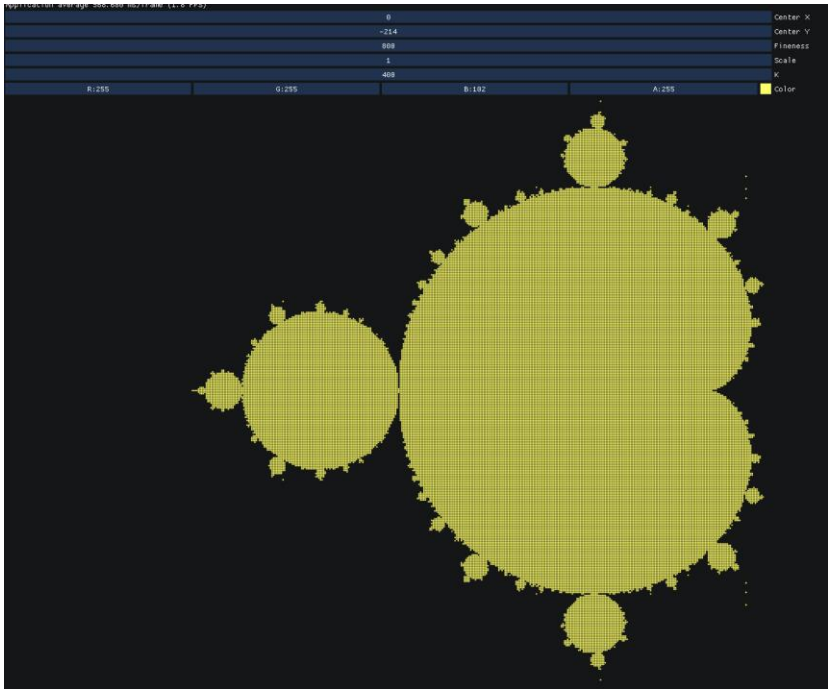

Figure 8. GUI Output for size=800, k=400.

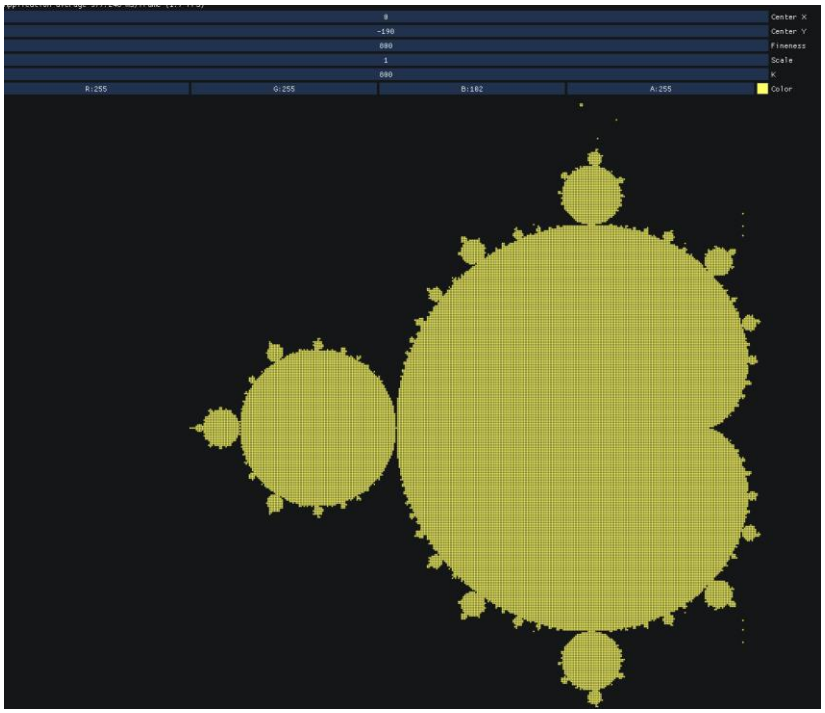

Figure 9. GUI Output for size=800, k=800.

		NUMBER OF PROCESSES					
		1	2	4	8	16	32
K	100	7670	15286	30429	59600	112288	152598
	200	4166	8330	16642	32203	60785	80264
	400	2190	4378	8755	16930	31783	41040
	800	1125	2253	4505	8719	16479	20982

Table 1. Calculation speed (pixels/second) of the MPI version of the program with respect to different values of k with fixed size of 800 and batch size of 10

		NUMBER OF PROCESSES					
		1	2	4	8	16	32
SIZE	400	15210	30328	60421	116304	158981	161196
	800	7670	15286	30429	59600	112288	152598
	1600	3835	7673	15300	30553	59235	110365
	3200	1913	3820	7612	14979	29208	53535

Table 2. Calculation speed (pixels/second) of the MPI version of the program with respect to different values of size with fixed k of 100 and batch size of 10

		NUMBER OF PROCESSES					
		1	2	4	8	16	32
K	100	7613	14603	26983	55030	101935	137348
	200	4162	8118	16196	31575	57216	76404
	400	2192	4144	8367	16751	31124	39420
	800	1128	2190	4415	8566	16317	20624

Table 3. Calculation speed (pixels/second) of the pthread version of the program with respect to different values of k with fixed size of 800 and batch size of 10

		NUMBER OF PROCESSES					
		1	2	4	8	16	32
SIZE	400	15279	29086	53610	100062	142552	140592
	800	7613	14603	26983	55030	101935	137348
	1600	3811	6766	13071	26736	55668	98347
	3200	1875	3175	6438	13000	27643	50448

Table 4. Calculation speed (pixels/second) of the pthread version of the program with respect to different values of size with fixed k of 100 and batch size of 10

Note:

a. All tests were conducted on 10.26.1.30.

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

2. Analysis

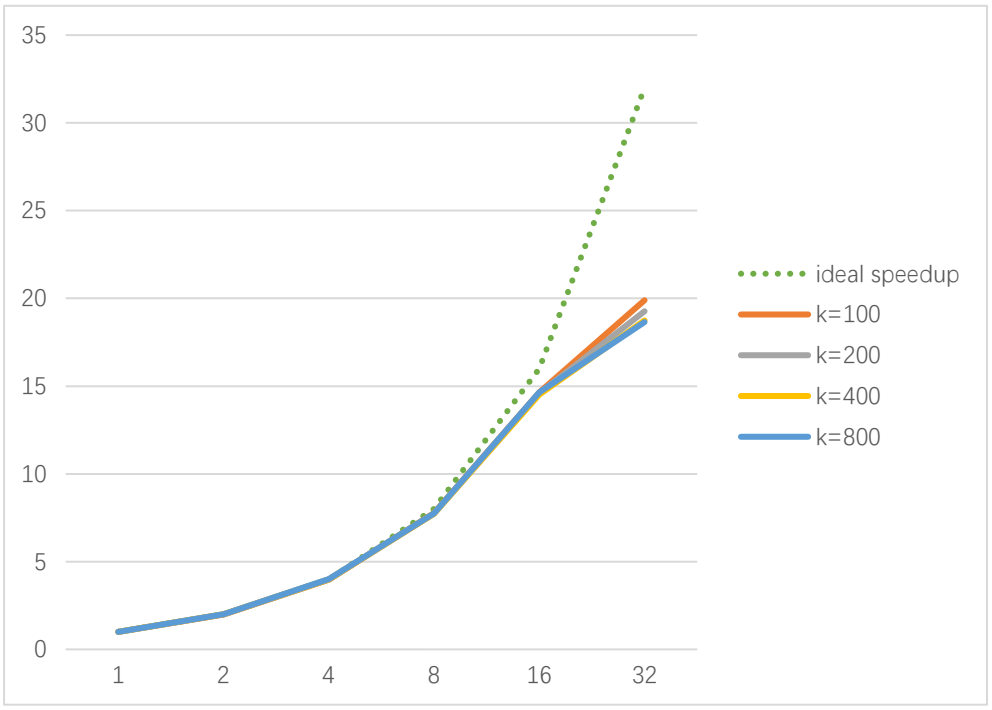

Figure 10. Calculation speed (pixels/second) of the MPI version of the program with respect to different values of k and different number of processes (with a fixed size of 800)

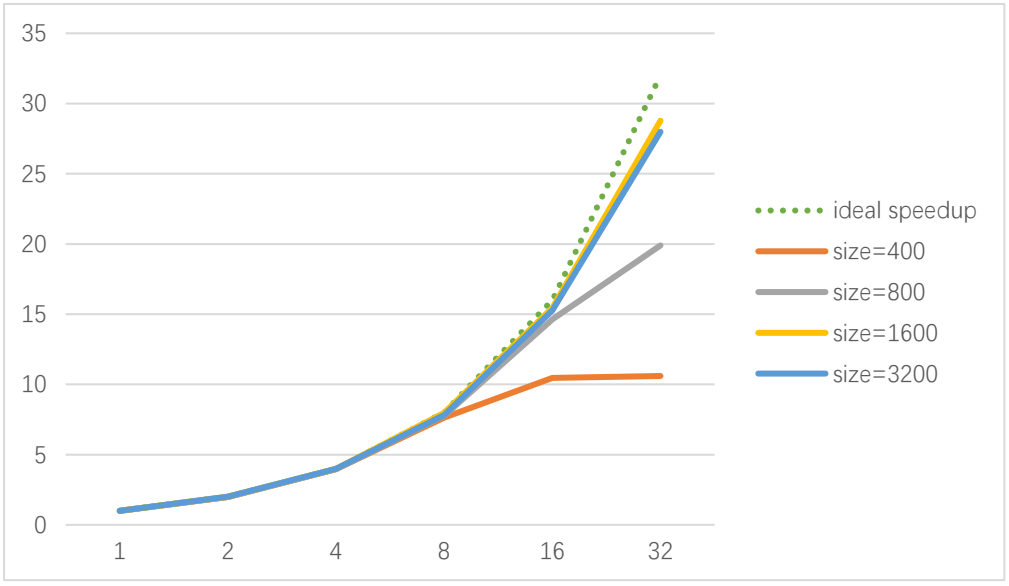

Figure 11. Calculation speed (pixels/second) of the MPI version of the program with respect to different values of size and different number of processes (with a fixed k of 100)

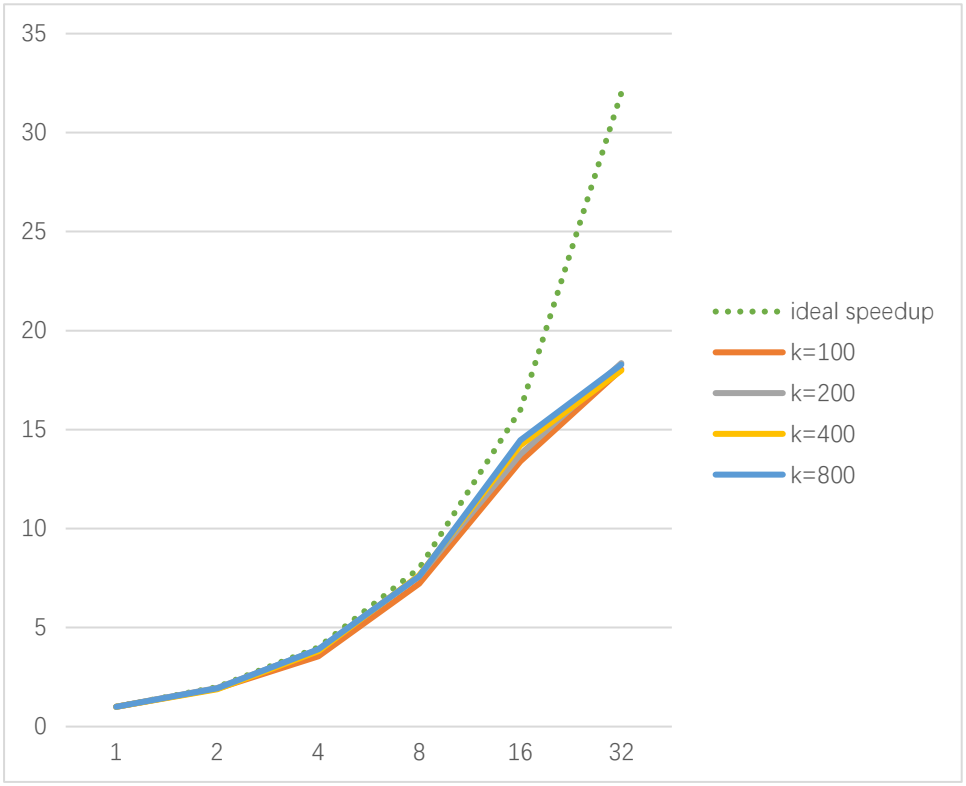

Figure 12. Calculation speed (pixels/second) of the pthread version of the program with respect to different values of k and different number of processes (with a fixed size of 800)

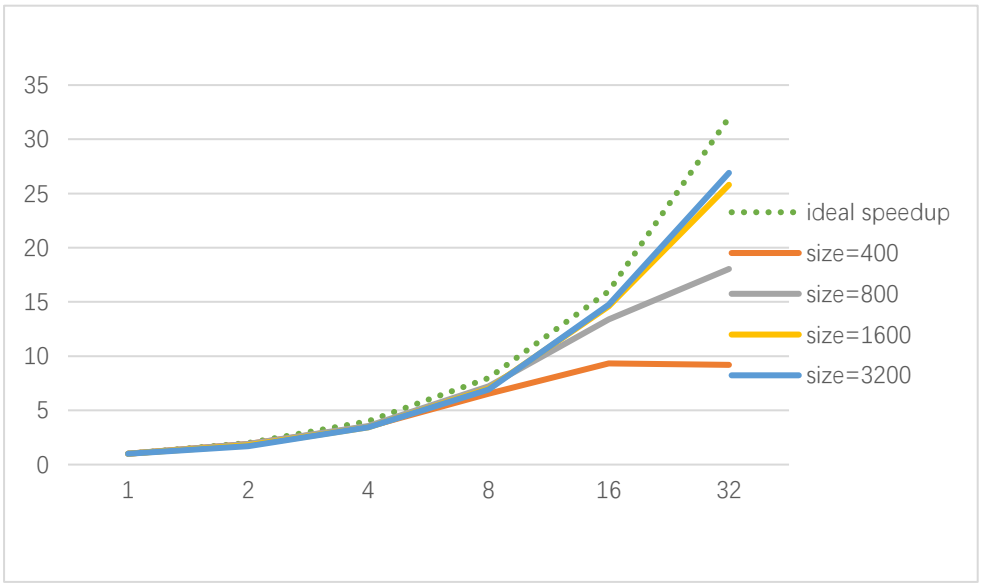

Figure 13. Calculation speed (pixels/second) of the pthread version of the program with respect to different values of size and different number of processes (with a fixed k of 100)

As shown in Figure 10 and Figure 12, the value of k has minor effect on the speedup factor in both MPI version and pthread version of the program. However, as in Figure 11 and Figure 13, the value of size has a significant impact on the speedup. When the size is small, the speedup is not good. Furthermore, when the number of processes/threads becomes large, the speedup almost stops to increase. The reason might be that the overhead of multiprocess/multithreading cancels its benefit when the size is small and the number of processes/threads is large. On the other hand, when the size is large, the speedup is better.

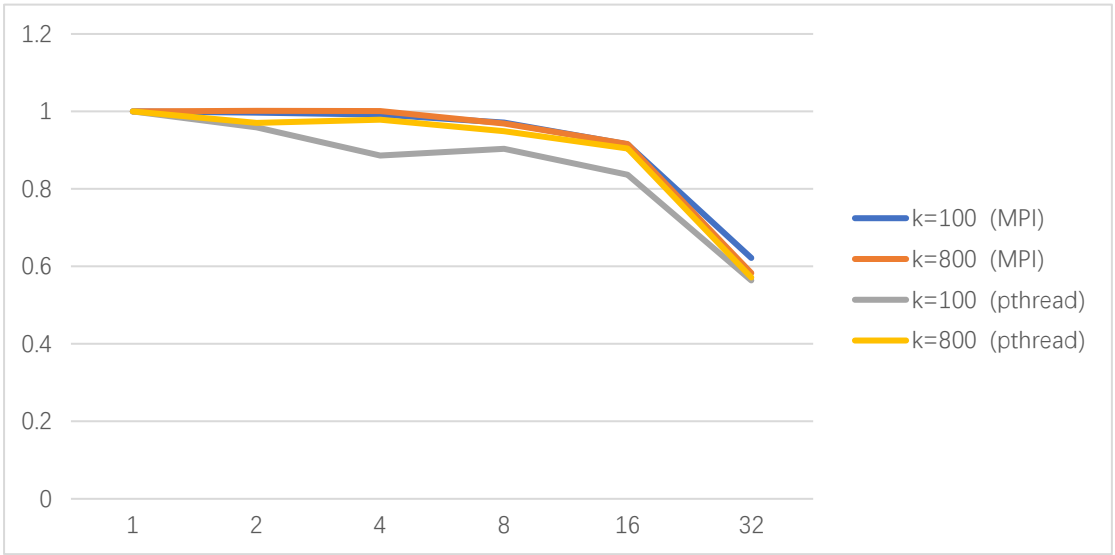

Figure 14. Efficiency of the program with respect to different values of k and different number of processes/threads (with a fixed size of 800)

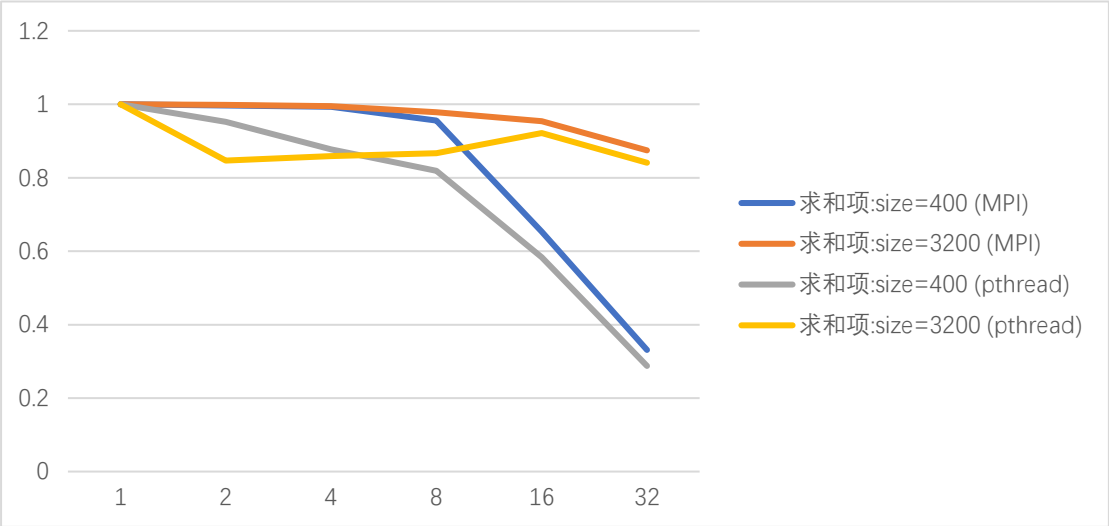

Figure 15. Efficiency of the program with respect to different values of k and different number of processes/threads (with a fixed size of 800)

Figure 14 and Figure 15 illustrate how efficiency change with the number of processes/threads under different configurations. Generally, the efficiency reduces as the number of processes/threads increases. The value of k does not have considerable impact on the efficiency, while the value of size affects the efficiency dramatically. Small sizes can lead to significant degradation in efficiency.

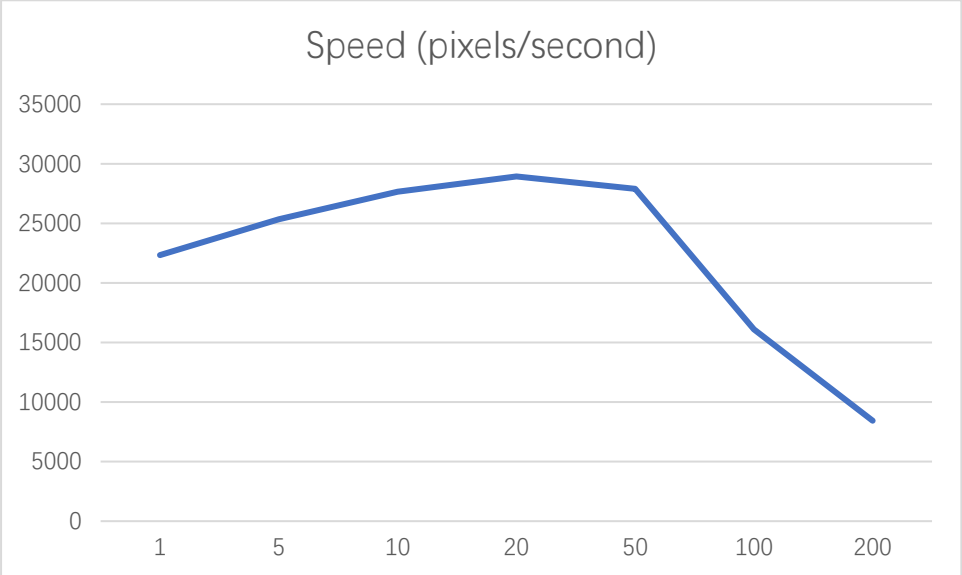

Figure 16. Calculation speed (pixels/second) of the pthread version of the program with respect to different values of batch size (with a fixed size of 3200, k of 100, and 16 threads)

As illustrated in Figure 14, the calculation speed is significantly affected by the chunk size (number of rows per chunk). For a fixed size of 3200, k of 100, and 16 threads, the optimal chunk size is between 10 and 50. Note that this optimal chunk size may not be true for other configurations. Theoretically, the best chunk size depends on size, k, and number of processes/values. When the chunk size is too small, the overhead of partitioning outweighs its benefit. When the chunk size is too large, the workloads of the processes/threads are unbalanced. Due to the Cask Effect, the speed becomes lower.

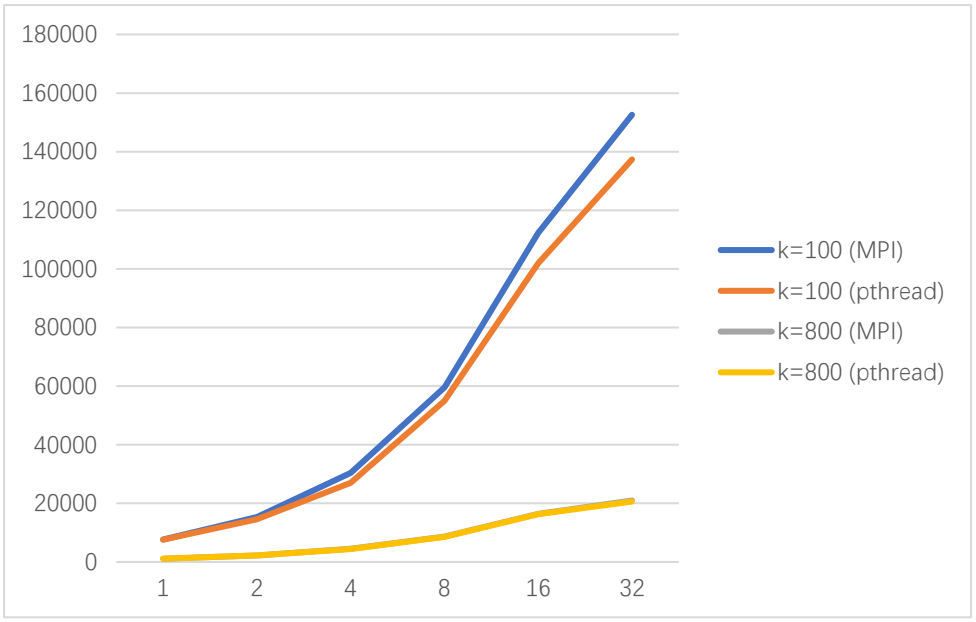

Figure 17. Calculation speed (pixels/second) of the MPI and pthread version of the program with (with a fixed size of 800)

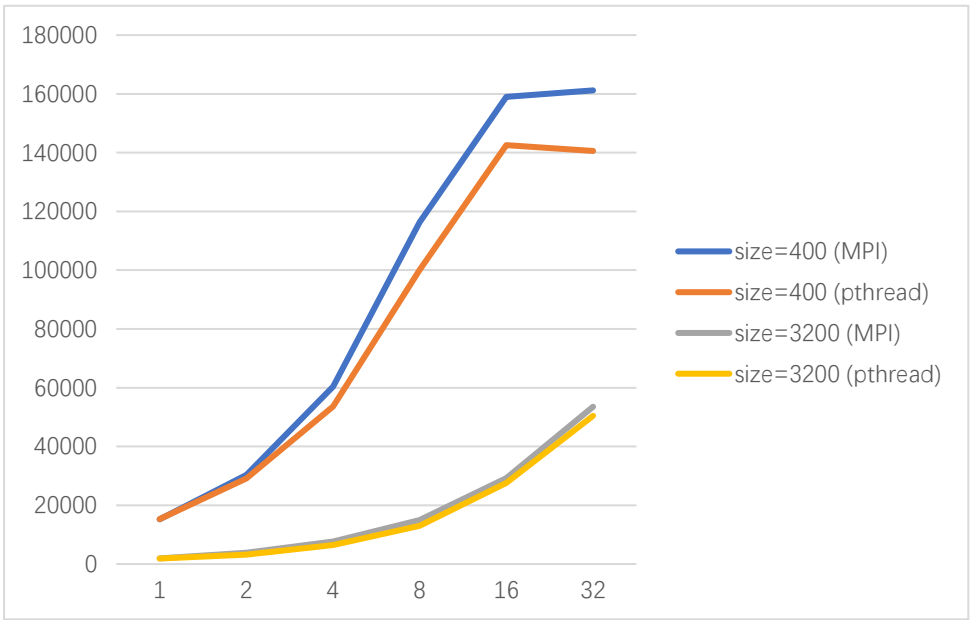

Figure 18. Calculation speed (pixels/second) of the MPI and pthread version of the program with (with a fixed k of 100)

The above two graphs compare the calculation speed of the MPI version and pthread version of the program. For large values of k and size, the

two versions show slight differences in performance. However, in terms of small values of k and size, the MPI version performs considerably better. This does not necessarily mean the MPI framework is better or more suitable for this task. More likely, this performance difference comes from my implementation of the program.

V. Conclusion

In summary, this assignment explores the Mandelbrot Set algorithm, the concepts and implementation of parallel computing, the MPI as well as pthread framework, the speedup and overhead of multiprocessing, and the basic GUI operations. In both MIPS and pthread versions of the program, due to the inter-process communication overhead, when the number of processes becomes larger, the efficiency of multiprocessing gets lower. For small sizes, this overhead may even cancel the speedup from parallelism when the number of processes is large. However, for large sizes, the impact of the communication overhead is relatively smaller since the portion of computation is larger. Therefore, parallel computing is best suitable for large sizes. On the other hand, the value of k does not affect the speedup significantly.

Moreover, the calculation speed is significantly affected by the chunk size (number of rows per chunk). When the chunk size is too small,

the overhead of partitioning outweighs its benefit. When the chunk size is too large, the workloads of the processes/threads are unbalanced. Due to the Cask Effect, the speed becomes lower.

There exist some limitations in this assignment. First, 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/multithreading will be close to 1. Also, we can infer that the multiprocessing/ multithreading 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 <chrono>
#include <iostream>
#include <graphic/graphic.hpp>
#include <imgui_impl_sdl.h>
#include <vector>
#include <complex>
#include <mpi.h>
#include <cstring>
#include <cstdint>
#include <cstddef>
#include <memory>

struct Square {
    std::vector<int> buffer;
    size_t length;

    explicit Square(size_t length) : buffer(length), length(length *
length) {}

    void resize(size_t new_length) {
        buffer.assign(new_length * new_length, false);
        length = new_length;
    }

    auto& operator[](std::pair<size_t, size_t> pos) {
        return buffer[pos.second * length + pos.first];
    }
};

void calculate(int* global_buffer, int size, int scale, double
x_center, double y_center, int k_value) {
    double cx = static_cast<double>(size) / 2 + x_center;
    double cy = static_cast<double>(size) / 2 + y_center;
    double zoom_factor = static_cast<double>(size) / 4 * scale;
    for (int i = 0; i < size; ++i) {
        for (int j = 0; j < size; ++j) {
            double x = (static_cast<double>(j) - cx) / zoom_factor;
            double y = (static_cast<double>(i) - cy) / zoom_factor;
            std::complex<double> z{0, 0};
            std::complex<double> c{x, y};
```

```
            int k = 0;
            do {
                z = z * z + c;
                k++;
            } while (norm(z) < 2.0 && k < k_value);
            global_buffer[i * size + j] = k;
        }
    }
}

void MPI_calculate(int* local_buffer, int size, int scale, double
x_center,
        double y_center, int k_value, int row_number, int chunk_size) {

    if (size == 0 || scale == 0) {
        return;
    }

    double cx = static_cast<double>(size) / 2 + x_center;
    double cy = static_cast<double>(size) / 2 + y_center;
    double zoom_factor = static_cast<double>(size) / 4 * scale;

    int i_max = size;
    if (row_number + chunk_size < size) {
        i_max = row_number + chunk_size;
    }

    int count = 0;
    for (int i = row_number; i < i_max; ++i) {
        for (int j = 0; j < size; ++j) {
            double x = (static_cast<double>(j) - cx) / zoom_factor;
            double y = (static_cast<double>(i) - cy) / zoom_factor;
            std::complex<double> z{0, 0};
            std::complex<double> c{x, y};
            int k = 0;
            do {
                z = z * z + c;
                k++;
            } while (norm(z) < 2.0 && k < k_value);
            local_buffer[count++] = k;
        }
    }
}
```

```
static constexpr float MARGIN = 4.0f;
static constexpr float BASE_SPACING = 2000.0f;
static constexpr size_t SHOW_THRESHOLD = 500000000ULL;


int main(int argc, char **argv) {
    MPI_Init(&argc, &argv);
    int rank;
    int num_proc;
    int points_per_proc;
    int* global_buffer;
    int* local_buffer;
    int* int_params_buffer;
    double* double_params_buffer;
    int chunk_size = 10;
    int global_row_number;
    int_params_buffer = new int[4];
    double_params_buffer = new double[2];
    int center_x = 0;
    int center_y = 0;
    int size = 800;
    int scale = 1;
    int k_value = 100;

    if (argc > 2) {
        size = atoi(argv[1]);
        k_value = atoi(argv[2]);
    }
    if (argc > 3) {
        chunk_size = atoi(argv[3]);
    }

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &num_proc);

    if (0 == rank) {
        graphic::GraphicContext context{"Assignment 2"};
        size_t duration = 0;
        size_t pixels = 0;
        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);
                static ImVec4 col = ImVec4(1.0f, 1.0f, 0.4f, 1.0f);
                ImGui::DragInt("Center X", &center_x, 1, -4 * size, 4 *
size, "%d");
                ImGui::DragInt("Center Y", &center_y, 1, -4 * size, 4 *
size, "%d");
                ImGui::DragInt("Fineness", &size, 10, 100, 1000, "%d");
                ImGui::DragInt("Scale", &scale, 1, 1, 100, "%.01f");
                ImGui::DragInt("K", &k_value, 1, 100, 1000, "%d");
                ImGui::ColorEdit4("Color", &col.x);
                {
                    using namespace std::chrono;
                    auto spacing = BASE_SPACING /
static_cast<float>(size);
                    auto radius = spacing / 2;
                    const ImVec2 p = ImGui::GetCursorScreenPos();
                    const ImU32 col32 = ImColor(col);
                    float x = p.x + MARGIN, y = p.y + MARGIN;
                    auto begin = high_resolution_clock::now();
                    global_buffer = new int[size * size + size];
                    if (num_proc == 1) {
                        calculate(global_buffer, size, scale, center_x,
center_y, k_value);
                    }
                    else {
                        int_params_buffer[0] = k_value;
                        int_params_buffer[1] = scale;
                        int_params_buffer[2] = size;
                        int_params_buffer[3] = chunk_size;
                        double_params_buffer[0] = center_x;
                        double_params_buffer[1] = center_y;
                        MPI_Bcast(int_params_buffer, 4, MPI_INT, 0,
MPI_COMM_WORLD);
                        MPI_Bcast(double_params_buffer, 2, MPI_DOUBLE, 0,
MPI_COMM_WORLD);
```

```
                        int finished_count = 0;
                        int row_offset;
                        int current_row = 0;

                        while (finished_count < size) {
                            MPI_Status status;
                            MPI_Recv(&row_offset, 1, MPI_INT,
MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, &status);
                            if (row_offset != -1) {
                                MPI_Recv(global_buffer + row_offset *
size,
                                         chunk_size * size, MPI_INT,
status.MPI_SOURCE, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
                                finished_count += chunk_size;
                            }
                            if (current_row < size) {
                                MPI_Send(&current_row, 1, MPI_INT,
status.MPI_SOURCE, 0, MPI_COMM_WORLD);
                                current_row += chunk_size;
                            }
                            else {
                                int code = -1;
                                MPI_Send(&code, 1, MPI_INT,
status.MPI_SOURCE, 0, MPI_COMM_WORLD);
                            }
                        }
                    }
                    auto end = high_resolution_clock::now();
                    pixels += size;
                    duration += duration_cast<nanoseconds>(end -
begin).count();
                    if (duration > SHOW_THRESHOLD) {
                        std::cout << pixels << " pixels in last " <<
duration << " nanoseconds\n";
                        auto speed = static_cast<double>(pixels) /
static_cast<double>(duration) * 1e9;
                        std::cout << "speed: " << speed << " pixels per
second" << std::endl;
                        pixels = 0;
                        duration = 0;
                    }
                    for (int i = 0; i < size; ++i) {
                        for (int j = 0; j < size; ++j) {
                            if (global_buffer[i * size + j] == k_value) {
```

```
                                draw_list->AddCircleFilled(ImVec2(x, y),
radius, col32);
                            }
                            x += spacing;
                        }
                        y += spacing;
                        x = p.x + MARGIN;
                    }
                    delete[] global_buffer;
                }
                ImGui::End();
            }
        });
    }
    else {
        while (true) {
            MPI_Bcast(int_params_buffer, 4, MPI_INT, 0, MPI_COMM_WORLD);
            MPI_Bcast(double_params_buffer, 2, MPI_DOUBLE, 0,
MPI_COMM_WORLD);
            int buffer_size = int_params_buffer[2] *
int_params_buffer[3];
            int row_offset = -1;
            MPI_Send(&row_offset, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
            do {
                MPI_Recv(&row_offset, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
                if (row_offset != -1) {
                    local_buffer = new int[buffer_size];
                    MPI_calculate(local_buffer, int_params_buffer[2],
int_params_buffer[1], double_params_buffer[0],
                        double_params_buffer[1], int_params_buffer[0],
row_offset, int_params_buffer[3]);
                    MPI_Send(&row_offset, 1, MPI_INT, 0, 0,
MPI_COMM_WORLD);
                    MPI_Send(local_buffer, buffer_size, MPI_INT, 0, 0,
MPI_COMM_WORLD);
                    delete[] local_buffer;
                }
            } while (row_offset != -1);
        }
    }
    delete[] int_params_buffer;
    delete[] double_params_buffer;
    MPI_Finalize();
```

```
    return 0;
}
```

2. Pthread Version

```
#include <chrono>
#include <iostream>
#include <graphic/graphic.hpp>
#include <imgui_impl_sdl.h>
#include <vector>
#include <complex>
#include <mpi.h>
#include <cstring>

struct Square {
    std::vector<int> buffer;
    size_t length;

    explicit Square(size_t length) : buffer(length), length(length *
length) {}

    void resize(size_t new_length) {
        buffer.assign(new_length * new_length, false);
        length = new_length;
    }

    auto& operator[](std::pair<size_t, size_t> pos) {
        return buffer[pos.second * length + pos.first];
    }
};

void calculate(Square &buffer, int size, int scale, double x_center,
double y_center, int k_value) {
    double cx = static_cast<double>(size) / 2 + x_center;
    double cy = static_cast<double>(size) / 2 + y_center;
    double zoom_factor = static_cast<double>(size) / 4 * scale;
    for (int i = 0; i < size; ++i) {
        for (int j = 0; j < size; ++j) {
            double x = (static_cast<double>(j) - cx) / zoom_factor;
            double y = (static_cast<double>(i) - cy) / zoom_factor;
            std::complex<double> z{0, 0};
```

```
            std::complex<double> c{x, y};
            int k = 0;
            do {
                z = z * z + c;
                k++;
            } while (norm(z) < 2.0 && k < k_value);
            buffer[{i, j}] = k;
        }
    }
}

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_barrier_t barrier_begin;
pthread_barrier_t barrier_end;
int chunk_size = 10;
int global_row_number;

void pthread_calculate(Square &buffer, int size, int scale, double
x_center, double y_center, int k_value, int row_number) {

    if (size == 0 || scale == 0) {
        return;
    }

    double cx = static_cast<double>(size) / 2 + x_center;
    double cy = static_cast<double>(size) / 2 + y_center;
    double zoom_factor = static_cast<double>(size) / 4 * scale;

    int i_max = size;
    if (row_number + chunk_size < size) {
        i_max = row_number + chunk_size;
    }

    for (int i = row_number; i < i_max; ++i) {
        for (int j = 0; j < size; ++j) {
            double x = (static_cast<double>(j) - cx) / zoom_factor;
            double y = (static_cast<double>(i) - cy) / zoom_factor;
            std::complex<double> z{0, 0};
            std::complex<double> c{x, y};
            int k = 0;
            do {
                z = z * z + c;
                k++;
            } while (norm(z) < 2.0 && k < k_value);
```

```
            buffer[{i, j}] = k;
        }
    }
}

static constexpr float MARGIN = 4.0f;
static constexpr float BASE_SPACING = 2000.0f;
static constexpr size_t SHOW_THRESHOLD = 500000000ULL;
int center_x = 0;
int center_y = 0;
int size = 800;
int scale = 1;
int k_value = 100;
Square canvas(100);

void* gui_loop(void* t) {
    int* int_ptr = (int *) t;
    int tid = *int_ptr;
    if (0 == tid) {
        graphic::GraphicContext context{"Assignment 2"};
        size_t duration = 0;
        size_t pixels = 0;
        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);
                static ImVec4 col = ImVec4(1.0f, 1.0f, 0.4f, 1.0f);
                ImGui::DragInt("Center X", &center_x, 1, -4 * size, 4 *
size, "%d");
                ImGui::DragInt("Center Y", &center_y, 1, -4 * size, 4 *
size, "%d");
                ImGui::DragInt("Fineness", &size, 10, 100, 1000, "%d");
                ImGui::DragInt("Scale", &scale, 1, 1, 100, "%.01f");
```

```
                ImGui::DragInt("K", &k_value, 1, 100, 1000, "%d");
                ImGui::ColorEdit4("Color", &col.x);
                {
                    using namespace std::chrono;
                    auto spacing = BASE_SPACING /
static_cast<float>(size);
                    auto radius = spacing / 2;
                    const ImVec2 p = ImGui::GetCursorScreenPos();
                    const ImU32 col32 = ImColor(col);
                    float x = p.x + MARGIN, y = p.y + MARGIN;
                    canvas.resize(size);
                    auto begin = high_resolution_clock::now();
                    pthread_mutex_lock(&mutex);
                    global_row_number = 0;
                    pthread_mutex_unlock(&mutex);
                    pthread_barrier_wait(&barrier_begin);
                    int row_number;
                    do {
                        pthread_mutex_lock(&mutex);
                        row_number = global_row_number;
                        if (row_number < size) {
                            global_row_number += chunk_size;
                        }
                        pthread_mutex_unlock(&mutex);
                        if (row_number < size) {
                            pthread_calculate(canvas, size, scale,
center_x, center_y, k_value, row_number);
                        }
                    } while (row_number + chunk_size < size);
                    pthread_barrier_wait(&barrier_end);
                    auto end = high_resolution_clock::now();
                    pixels += size;
                    duration += duration_cast<nanoseconds>(end -
begin).count();
                    if (duration > SHOW_THRESHOLD) {
                        std::cout << pixels << " pixels in last " <<
duration << " nanoseconds\n";
                        auto speed = static_cast<double>(pixels) /
static_cast<double>(duration) * 1e9;
                        std::cout << "speed: " << speed << " pixels per
second" << std::endl;
                        pixels = 0;
                        duration = 0;
                    }
```

```
                    for (int i = 0; i < size; ++i) {
                        for (int j = 0; j < size; ++j) {
                            if (canvas[{i, j}] == k_value) {
                                draw_list->AddCircleFilled(ImVec2(x, y),
radius, col32);
                            }
                            x += spacing;
                        }
                        y += spacing;
                        x = p.x + MARGIN;
                    }
                }
                ImGui::End();
            }
        });
    }
    else {
        while (true) {
            pthread_barrier_wait(&barrier_begin);
            int row_number;
            do {
                pthread_mutex_lock(&mutex);
                row_number = global_row_number;
                if (row_number < size) {
                    global_row_number += chunk_size;
                }
                pthread_mutex_unlock(&mutex);
                if (row_number < size) {
                    pthread_calculate(canvas, size, scale, center_x,
center_y, k_value, row_number);
                }
            } while (row_number + chunk_size < size);
            pthread_barrier_wait(&barrier_end);
        }
    }
    return nullptr;
}

int main(int argc, char **argv) {
    if (argc < 2) {
        exit(1);
    }
    if (argc > 3) {
        size = atoi(argv[2]);
```

```
        k_value = atoi(argv[3]);
    }
    if (argc > 4) {
        chunk_size = atoi(argv[4]);
    }
    int num_threads = atoi(argv[1]);
    pthread_barrier_init(&barrier_begin, nullptr, num_threads);
    pthread_barrier_init(&barrier_end, 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;
}
```