CSC4005HW2: Mandelbrot Set Computation & Analysis

118010202 / LIU Zhixuan

November 1, 2021

1 Abstract

The Mandelbrot set is the set of complex numbers c for which the function $f_c(z) = z^2+c$ does not diverge to infinity when iterated from $z = 0$. Images of the Mandelbrot set exhibit an elaborate and infinitely complicated boundary that reveals progressively ever-finer recursive detail at increasing magnifications, making the boundary of the Mandelbrot set a fractal curve.

In this assignment, I performed both parallel version and sequential version to draw the Mandelbrot Set using ImGui; moreover, for the parallel implementation, I use both MPI method and pthread method to realize this function. At last, the corresponding experiments and some efficient analysis is performed on all the algorithms.

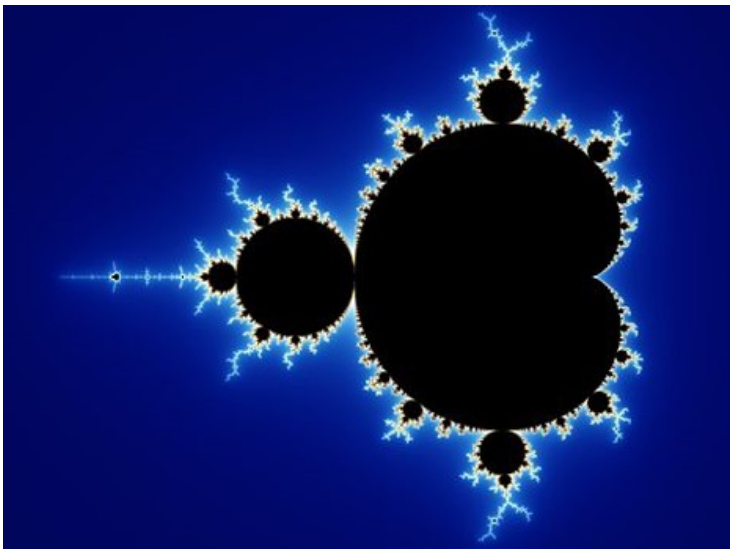

Figure 1: Illustration of Mandelbrot Set Computation

2 Introduction to Mandelbrot Set

In this section, we briefly introduce Mandelbrot Set computation and how to implement the sequential version of it using C++ and some complexity analysis of it.

Mandelbrot set is a set of points in a complex number giving the position of the point in the complex plane. All the points are quasistable when computed by iterating the function:

$$z_{k+1} = z_k^2 + c$$

Where $z_k + 1$ is the $(k+1)$th iteration of the complex number $z = a + b_i$, z_k is the kth iteration of z, and c is a complex number giving the position of the point in the complex plane. The initial value for z is zero and the iteration are continues until the magnitude of z is greater than 2 or the number of iteration reaches some arbitrary limitation.

Drawing the Mandelbrot Set is like a bit-wise drawing. The resolution is defined by the user, therefore, if the the resolution is set to be higher, the sample rate will be higher, and there will be much point calculated in the figure; if the resolution is set to be smaller, than there will be less sample in the picture, therefore, the drawing is discerned to be more discrete.

Now, let's talk about the code implementation of this sequential algorithm using C++. In the following codes, i use ImGui to implement the drawing process.

Firstly, I define a buffer, which is called canvas, to store the information of the synthesis drawings.

```
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];
    }
};
Square canvas(100);
```

Then, we set some important parameters of the whole picture. These parameters can be set during the ImGui running process.

```
static int center_x = 0;
static int center_y = 0;
static int size = 800;
static int scale = 1;
static ImVec4 col = ImVec4(1.0f, 1.0f, 0.4f, 1.0f);
static int k_value = 100;
```

And the last step is to calculate the Mandelbrot Set pixel by pixel and write the information in the canvas buffer, and then draw the things in the canvas buffer using ImGui.

Basically, the main idea to draw the things is to first construct the complex number c, and then iterate using the computation mentioned above. If it converge within the set k-value, than we will mark this complex number based on its corresponding x and y axis.

```
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;
    }
}
```

After writing the information into canvas, the final step is to use ImGui to draw the canvas.

Now, let's evaluate the complexity of sequential Mandelbrot Set computation algorithm. The computational complexity : $O(N \times M)$ where, N and M are the size of the drawing (in other the words, the resolution), and in our cases, we often set $M = N$, therefore, the computational complexity of our cases are $O(N^2)$. This is because to finally draw the Mandelbrot Set, the computation is pixel wise, and for each pixel, the computation is almost the same.

3 Parallel Mandelbrot Set Computation using MPI

It is obvious that sequential Mandelbrot Set computation does not have a very good performance in time complexity. However, this algorithm can be much more efficient if it is distributed computed, therefore, the computational time will be saved. The first idea of doing this is to use MPI method to spead up the process, and the main idea is as follows:

In the MPI algorithm, firstly, the assignment will be given and initialized in the master process. Then the master process will distribute the information in the slave processes, each slave process is equally assigned to on part of the whole picture, and their aim is to draw their own part of the picture. After drawing their own parts, they will sent the result to the master process. And the master process will write the information in the buffer "canvas", and then using ImGui to visualize the drawing.

The process for this distributed computing is shown in the figure 3 below:

Now let's look at how to implement this process using C++.

Similar to the sequential process, we first initiate a structure called canvas, which stored the information of the drawing based on the pixel in the master process, and initialize some necessary parameters.

Next, in the MASTER process, the MASTER will send the parameter information to all the slave process by using iteration:

```
for (int m=0; m<slave_size; m++){
        MPI_Isend(&info, 5, MPI_INT, (m+1), 0, MPI_COMM_WORLD, &request);
    }
```

Then, in the slave process, where all the computations are distributed, slave first receieve all the information sent by MASTER process:

```
MPI_Recv(&re_info, 5, MPI_INT,0,0,MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

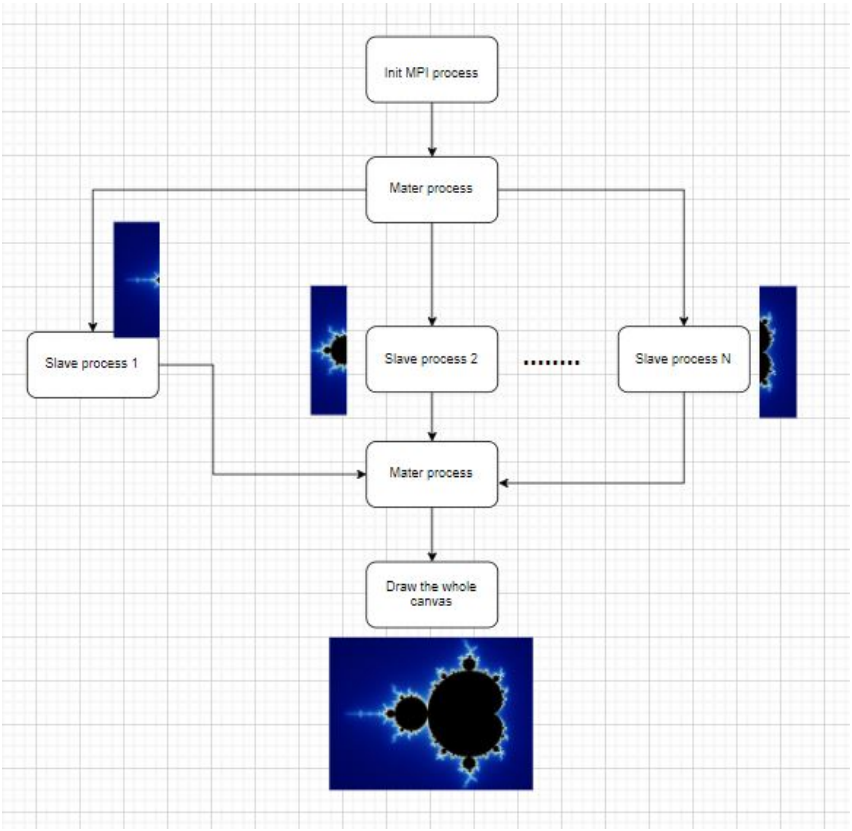

Figure 2: Process of Parallel computation using MPI

Then, only calculate part of the image and store the image information in the local buffer:

```
void calculate(int *buffer, int size, int len, int scale, double x_center, double
    y_center, int k_value, int rank) {
    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 < len; ++i) {
        for (int j = 0; j < size; ++j) {
            double x = (static_cast<double>(j) - cx) / zoom_factor;
            double y = (static_cast<double>(i) + (rank-1)*len - 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*size + j] = k;
        }
    }
}
```

After calculate their own part, the slave will send the buffer to the MASTER process, and the slave process finishes its job.

```
MPI_Isend(&(local_result[0]), local_size*s_size, MPI_INT, 0,1, MPI_COMM_WORLD, &
    request);
```

Then, the MASTER process receive the buffer from the slave processes, and write the information to the canvas:

```
MPI_Recv(&(imm_result[0]), local_length*size, MPI_INT,MPI_ANY_SOURCE, 1,
    MPI_COMM_WORLD,&status);
    slave_rank = status.MPI_SOURCE;
    int base_index = (slave_rank-1)*local_length;

    for(int n=0; n<local_length; n++){
        for(int q=0; q<size; q++){
            canvas[{base_index+n,q}] = imm_result[n*size+q];
        }
    }
```

The last part is for canvas to draw the information using ImGui:

```
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;
}
```

This is pretty much about the concise implementation of the parallel Mandekbrot Set Computation using MPI.

As for the computational complexity, since the whole mathematical computational algorithm does not change, the only change is that the whole process is distributed to p slave processes to compute, therefore, the computational complexity can be written as $O(\frac{N\times M}{p})$, or in our cases are $O(\frac{N^2}{p})$.

4 Parallel Mandelbrot Set Computation using Pthread

It is obvious that sequential Mandelbrot Set computation does not have a very good performance in time complexity. However, this algorithm can be much more efficient if it is distributed computed, therefore, the computational time will be saved. The second idea of doing this is to use Pthread method to spead up the process, and the main idea is as follows:

In the Pthread algorithm, firstly, the assignment will be given and initialized in the main function. Then the main function will fork several child processes, and will distribute the information in the child processes, each child process is equally assigned to on part of the whole picture, and their aim is to draw their own part of the picture. Each child process will write the information in the buffer "canvas". After writing the information, the each thread will join together, and then using ImGui to visualize the drawing.

The process for this distributed computing is shown in the figure 3 below:

Now let's look at how to implement this Pthread process using C++.

Similar to the sequential process, we first initiate a structure called canvas, which stored the information of the drawing based on the pixel in the master process, and initialize some necessary parameters.

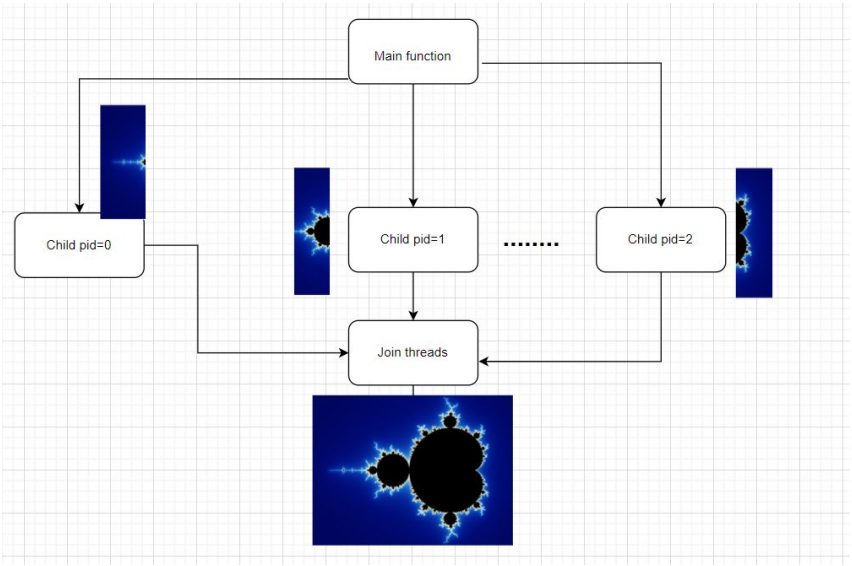

Figure 3: Process of Parallel computation using Pthread

Next, in the main function process, we will create a space to store some threads, based on the given threads number:

```
pthread_t threads[THREADS_NUM];
```

Then, we also create a structure called argument, which will save all the information that each thread's function will receive, and create and send the information to thread:

```
struct Arguments{
    Square *canvas;
    int size;
    int len;
    int scale;
    double x_center;
    double y_center;
    int k_value;
    int pid;
};

for(int m=0; m<THREADS_NUM; m++){
    pthread_create(&threads[m], nullptr, local_process, new Arguments{
        .canvas = &canvas,
        .size = size,
        .len = local_size,
        .scale = scale,
        .x_center = static_cast<double>(center_x),
        .y_center = static_cast<double>(center_y),
        .k_value = k_value,
        .pid = m
    });
}
```

Then, int the child process, firstly, they receive the information, and then only calculate part of the image and store the image information in the canvas. To let them all write to the canvas correctly, I passed the pointer of the canvas so that each thread can find the address to write to:

```
void *local_process(void *arg_ptr){
    auto arguments = static_cast<Arguments *>(arg_ptr);
    // CALCULATE(ARGUMENTS->CANVAS, ARGUMENTS->SIZE, ARGUMENTS->LEN, ARGUMENTS->
        SCALE, ARGUMENTS->X_CENTER, ARGUMENTS->Y_CENTER, ARGUMENTS->K_VALUE,
        ARGUMENTS->PID);
    double cx = static_cast<double>(arguments->size) / 2 + arguments->x_center;
    double cy = static_cast<double>(arguments->size) / 2 + arguments->y_center;
    double zoom_factor = static_cast<double>(arguments->size) / 4 * arguments->
        scale;
    for (int i = 0; i < arguments->len; ++i) {
        for (int j = 0; j < arguments->size; ++j) {
            double x = (static_cast<double>(j) - cx) / zoom_factor;
            double y = (static_cast<double>(i)+(arguments->pid)*arguments->len - 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 < arguments->k_value);
            (*arguments->canvas)[{i+(arguments->pid)*arguments->len, j}] = k;
        }
    }
    delete arguments;
    return nullptr;
}
```

After calculate their own part, the all the threads will join together, and the whole process ends.

```
for(auto & n : threads){
    pthread_join(n, nullptr);
}
pthread_attr_destroy(&attr);
```

The last part is for canvas to draw the information using ImGui:

```
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;
}
```

This is pretty much about the concise implementation of the parallel Mandekbrot Set Computation using Pthread.

As for the computational complexity, since the whole mathematical computational algorithm does not change, the only change is that the whole process is distributed to c child processes to compute, therefore, the computational complexity can be written as $O(\frac{N \times M}{c})$, or in our cases are $O(\frac{N^2}{c})$.

5 Compile and Run

I use the template on the blackboard to write the program, which use ImGui to display the image. So here are some command to run my code.

IMPORTANT: Please put my codes under file src and use the file name: "main.cpp". For example, put one of my file is: "A2_mpi.cpp", please change it to "main.cpp" and put it under src file in "csc4005_imgui" folder.

5.1 Build

For all the file, please do the following to build the file:

```
$ rm -rf build
$ mkdir build
$ cd build
$ cmake ..
$ make
```

5.2 Run Sequential Program

To run my sequential codes, the sbatch shell codes are as follows:

```
#!/bin/bash
#SBATCH -J Zhixuan
#SBATCH -o sequential.out
#SBATCH -N 1
#SBATCH -n 1
#SBATCH -t 1

xvfb-run -a mpirun -np 1 csc4005_imgui
```

5.3 Run MPI Program

To run my MPI codes, the sbatch shell codes are as follows:

```
#!/bin/bash
#SBATCH -J Zhixuan
#SBATCH -o MPI_N1n4.out
#SBATCH -N 1
#SBATCH -n 4
#SBATCH -t 1

xvfb-run -a mpirun -np 4 csc4005_imgui
```

5.4 Run Pthread Program

To run my Pthread codes, the sbatch shell codes are as follows:

```
#!/bin/bash
#SBATCH -J Zhixuan
#SBATCH -o Pthread.out
#SBATCH -N 1
#SBATCH -n 1
#SBATCH -t 1
```

```
7
8 xvfb-run -a mpirun -np 1 csc4005_imgui
```

To test different thread, please change the thread number in the first line of the code, where you can modify the predefined THREADS_NUM.

```
1  #include <chrono>
2  #include <iostream>
3  #include <graphic/graphic.hpp>
4  #include <imgui_impl_sdl.h>
5  #include <vector>
6  #include <complex>
7  #include <cstring>
8  #include <cmath>
9
10
11 pthread_mutex_t mutex;
12 pthread_cond_t mutex_threshold;
13 #define THREADS_NUM 4
14
```

Figure 4: Change the thread number here

5.5 Sample Outcomes

Here are some sample outcomes for both sequential, mpi, and pthread algorithms.

This figure 5 shows how terminal looks like:

```
52   800 pixels in last 565890179 nanoseconds
53   speed: 1413.7 pixels per second
54   800 pixels in last 566192420 nanoseconds
55   speed: 1412.95 pixels per second
56   800 pixels in last 565987280 nanoseconds
57   speed: 1413.46 pixels per second
58   800 pixels in last 566166229 nanoseconds
59   speed: 1413.01 pixels per second
60   800 pixels in last 565926241 nanoseconds
61   speed: 1413.61 pixels per second
62   800 pixels in last 566048631 nanoseconds
63   speed: 1413.31 pixels per second
64   800 pixels in last 566317295 nanoseconds
```

Figure 5: Outcomes for terminal

The following figures 6 7 8 9 10 shows some sample outcomes when user change the parameters (especially the size (or the resolution) of the image):

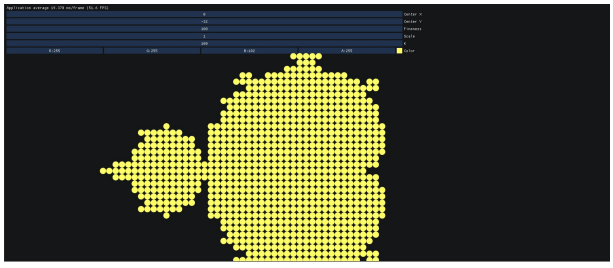

Figure 6: Outcomes when size is 100

And some raw datas:

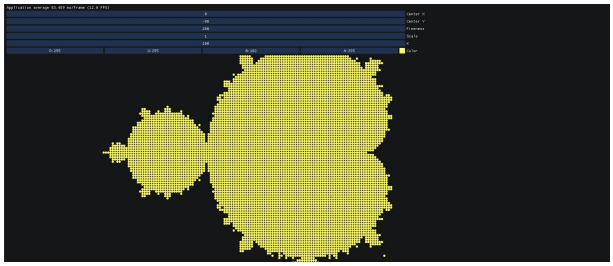

Figure 7: Outcomes when size is 200

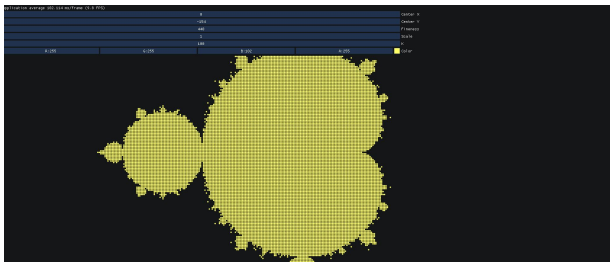

Figure 8: Outcomes when size is 440

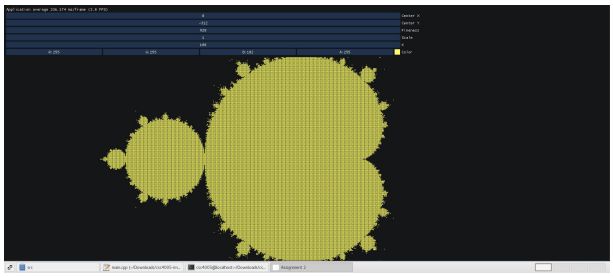

Figure 9: Outcomes when size is 920

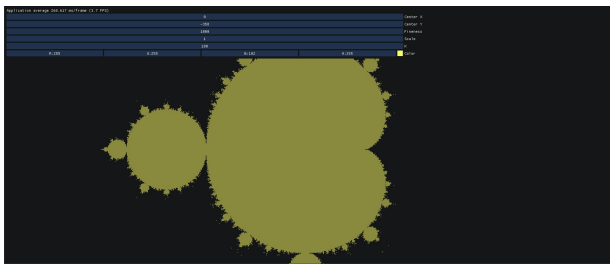

Figure 10: Outcomes when size is 1000

process=4	100	200	300	400	500	600	700	800	900	1000
seq	0.001234	0.002135	0.003577	0.014444	0.03273	0.058392	0.091696	0.129019	0.181891	0.235336
mpi	0.000353	0.00061	0.001022	0.004127	0.009351	0.016683	0.026199	0.036863	0.051969	0.067239
thread	0.000309	0.000534	0.000894	0.003611	0.008182	0.014598	0.022924	0.032255	0.045473	0.058834
process=8	100	200	300	400	500	600	700	800	900	1000
seq	0.001234	0.002135	0.003577	0.014444	0.03273	0.058392	0.091696	0.129019	0.181891	0.235336
mpi	0.000184	0.000319	0.000534	0.002156	0.00521	0.01	0.018	0.025	0.036	0.05
thread	0.000154	0.000267	0.000447	0.001805	0.004091	0.007299	0.011462	0.02	0.022736	0.029417

Figure 11: Time vs Job Size when process numbers for MPI and Pthread are fixed

6 Analysis

6.1 Comparison between MPI, Pthread and Sequential

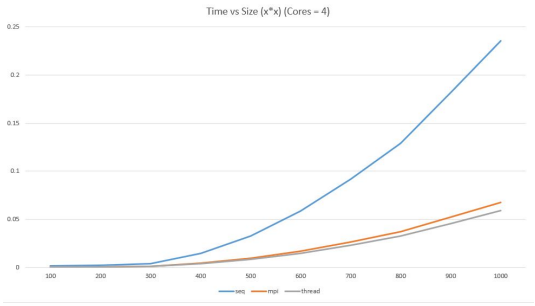

Figure 12: Execution time vs Job size when MPI and Pthread use 4 processer

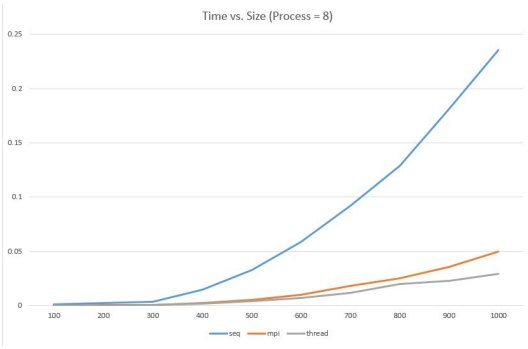

Figure 13: Execution time vs Job size when MPI and Pthread use 8 processer

As the figure shown above, when we fix the number of processes used by the program for MPI and Pthread (here we are talking about the fixed number, regardless of the number of masters in the mpi program, because when the MPI is executing the program, the MASTER program actually does not perform calculations, and It is to allocate tasks. In order to make the comparison more convincing, the number of processes mentioned here is slave process and child process). Adjust the job size of the program. The graph we get reflects the growth of time. In this figure, we can see that in the three programs, time has a growth trend similar to a quadratic function with the increase of job size. This is in line with our previous analysis that the computational complexity of the three algorithms is quadratic.

At the same time, we found that mpi and pthread programs are shorter in time than sequential programs, indicating that our distributed design can accelerate computing efficiency. We found that when the process number is 8, mpi and pthread accelerate the program more than when the process number is 8. This observation is consistent with the computational complexity of mpi and pthread mentioned earlier, and we will analyze the specific multiples later.

At the same time, we also found that although the time used by mpi is very close to that of pthread, it still takes more time than pthread. This may be because of my program design. Because in pthread, all child processes will be rewritten directly on the canvas, they do not need a local buffer to store data. But in my mpi algorithm, I use the local buffer to store the data of the slave processes and pass this data to the MASTER process. At the same time, the MATER process also needs to write the received signal into the canvas. This process is very Time-consuming. This point can explain why pthread is generally faster than mpi.

6.2 Time vs Process Number

The above three pictures show the relationship between the time spent by mpi and pthread and the process number when the job size is fixed. For the fairness of comparison, this type of process number refers to the slave process of mpi and the child process of pthread. We found that when the number of processes increases, the time has a downward trend. This

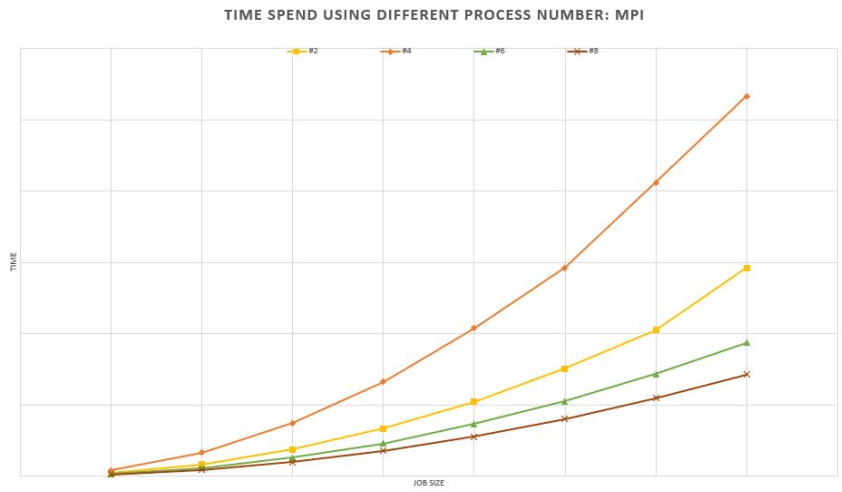

Figure 14: Time vs Job size in different process number in MPI

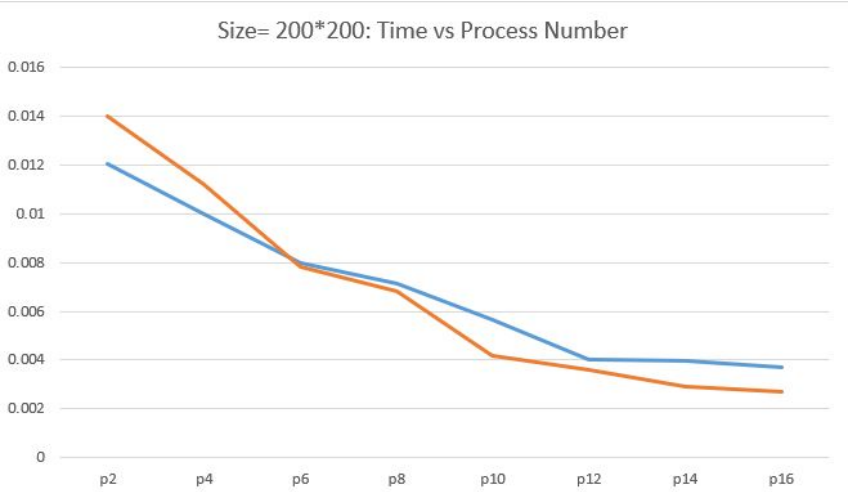

Figure 15: Time vs Process when JOB Size is fixed to be 200 for MPI and Pthread

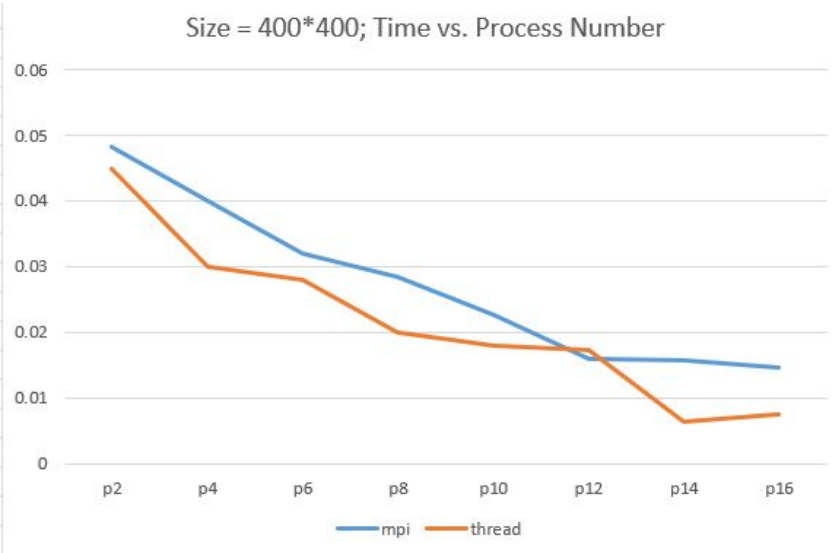

Figure 16: Time vs Process when JOB Size is fixed to be 400 for MPI and Pthread

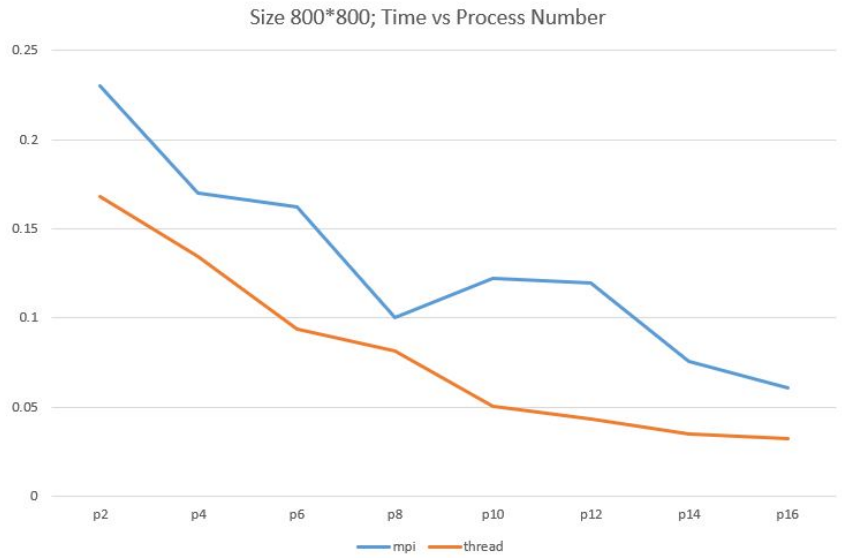

Figure 17: Time vs Process when JOB Size is fixed to be 800 for MPI and Pthread

trend is consistent with the computational complexity of the two previously analyzed. When the number of process numbers is larger, the amount of work allocated to each process will also become less, so the time taken to complete a single computing task will also become less. This effectively shows that distributed computing can reasonably increase the time used.

At the same time, we also found that although the time used by mpi is very close to that of pthread, it still takes more time than pthread within the tolerance range. This point may be the same as the reason we analyzed in the previous section. Because in pthread, all child processes will be rewritten directly on the canvas, they do not need a local buffer to store data. But in my mpi algorithm, I use the local buffer to store the data of the slave processes and pass this data to the MASTER process. At the same time, the MATER process also needs to write the received signal into the canvas. This process is very Time-consuming.

7 Conclusion and Future work

This project uses sequential and parallel Mandelbrot Set Computation, and for the parallel process, we use both MPI and pthread to experiment regarding their performance. I obvserved that if there are more processes, the acceleration overhead will increase. From the experiment, I found that when the process number is increased, parallel programs do provide faster performance.

Moreover, I also find that in the same condition, my implimentation of pthread is actually much faster than MPI. This is due to the face that in my practice, pthread directly write to the buffer which store the information of generated image while in MPI I need to use local buffer and then send the information to the master process, then the master process writes to the image buffer. This is actually time consuming point. And this point can be further improved.