# CSC4005 FA22 HW02

Haoran Sun (haoransun@link.cuhk.edu.cn)

## 1 Introduction

The Mandelbrot set is the set of complex numbers $c \in \mathbb{C}$ that does not diverges under the following iteration, start from $z_0 = c$.

$$z_{n+1}^2 = z^2 + c \tag{1}$$

The set can be visualized through numerical computation. By setting a rectangular region (should be aligned to the real and imaginary axis) in the complex plane, one can divide it into a mesh of points. Setting a maximum iteration number $m_{\max}$ and then perform iteration on each point using the equation 1 until it diverges. Let $m(c)$ be the number of iteration steps that the point diverges, then $m(c)/m_{\max}$ could be an estimator of the density of Mandelbrot set around this point. Then, the density of Mandelbrot set could be visualized according to $m(c)$ and $m_{\max}$.

However, the computation is intensive. Suppose we are going to obtain an image with resolution $m \times n$, the computational complexity would be $O(nm)$. In order to accelerate the computation, in this assignment, two parallel scheme are implemented: MPI and Pthread. Programs are tested under different resolutions and numbers of CPU cores. Speed-up factor and CPU efficiency are also analyzed.

## 2 Method

### 2.1 Program design and implementation

All programs are implemented using C++ programming language. MPICH and Pthreads are used for parallelism. For visualization, OpenGL is used to implement a graphical render while STB (single-file public domain libraries) is used to plot png images.

Please refer to Figure A.1 for the MPI program flowchart and Figure A.2 for the Pthreads program flowchart. The sequential version were written in `src/main.seq.cpp`, MPI version `src/main.mpi.cpp`, Pthreads `src/main.pthread.cpp`.

### 2.2 Usage

*For convenience, one can directly execute* `demo.seq.sh`*,* `demo.mpi.sh`*, and* `demo.pthread.sh` *under* `hw02/` *to have a first glimpse of the program.*

```
cd hw02
./scripts/demo.seq.sh
./scripts/demo.mpi.sh
./scripts/demo.pthread.sh
```

The program is compiled using CMake build system. One can have a look at `CMakeLists.txt` and `src/CMakeLists.txt` to check compilation requirements. If one wants to build the program with the GUI feature, he can run the following commands to configure and start compilation under hw02 directory. To disable the GUI feature, one can set -DGUI=OFF in the configure process. The compiled programs would be placed in build/bin directory.

```
cmake -B build -DCMAKE_BUILD_TYPE=Release -DGUI=ON
cmake --build build
```

One can run the program using the following commands, where xmax, xmin, ymax, ymin set the range of the rectangular region in the complex plane; ndim sets the resolution (partition of the mesh) on $x$-direction (real line); record determines whether the runtime data would be saved; save controls if the image would be saved. After executing the program, a GUI window should be prompted to display the density and an image mandelbrot_$jobtype.png similar to Figure 1 would be saved.

```
paras="--ndim 2000 --xmin -0.125 --xmax 0.125 --ymin 0.6 --ymax 0.7"
record="--record 0"
save="--save 1"
./build/bin/main.seq $paras $save $record              # sequential program
mpirun -np 4 ./build/bin/main.mpi $paras $save $record # mpi program
./build/bin/main.pthread -nt 4 $paras $save $record    # pthreads program
```

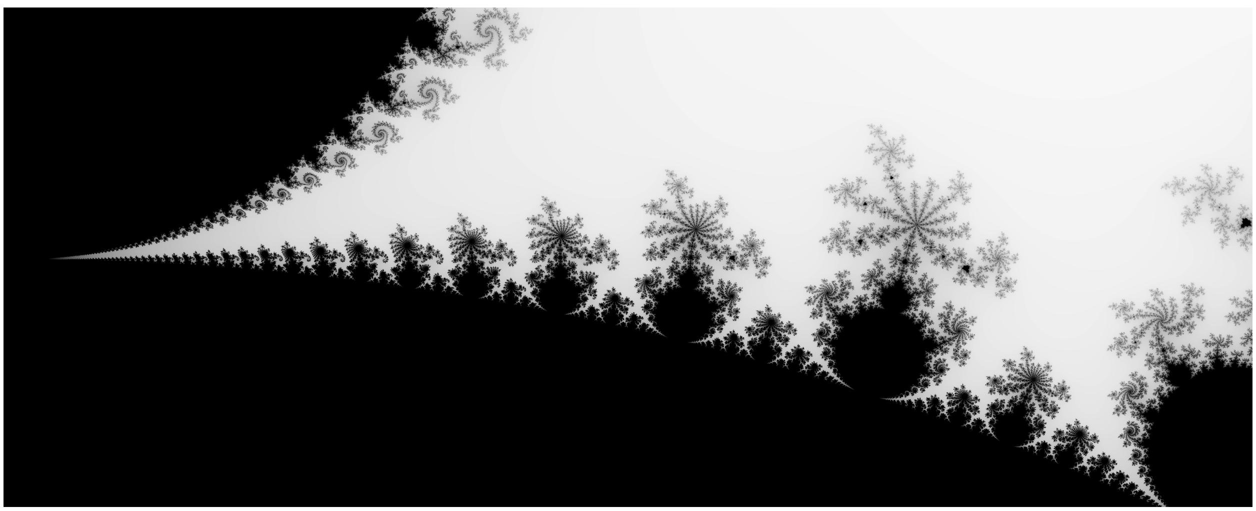

Figure 1: Sample output image

## 2.3 Performance evaluation

In order to evaluate the parallel code, the program was executed under different configurations. With 20 different CPU core numbers (from 1 to 20 with increment 1, $p = 1, 2, \dots, 20$) and 20 different $x$-resolutions (from 500 to 10000 with increment 500, $n = 500, 1000, \dots, 10000$), 400 cases in total were sampled. Recorded runtime and CPU time were analyzed through the Numpy package in Python. Figures were plotted through the Matplotlib and the Seaborn packages in Python. Analysis code were written in analysis/main.ipynb.

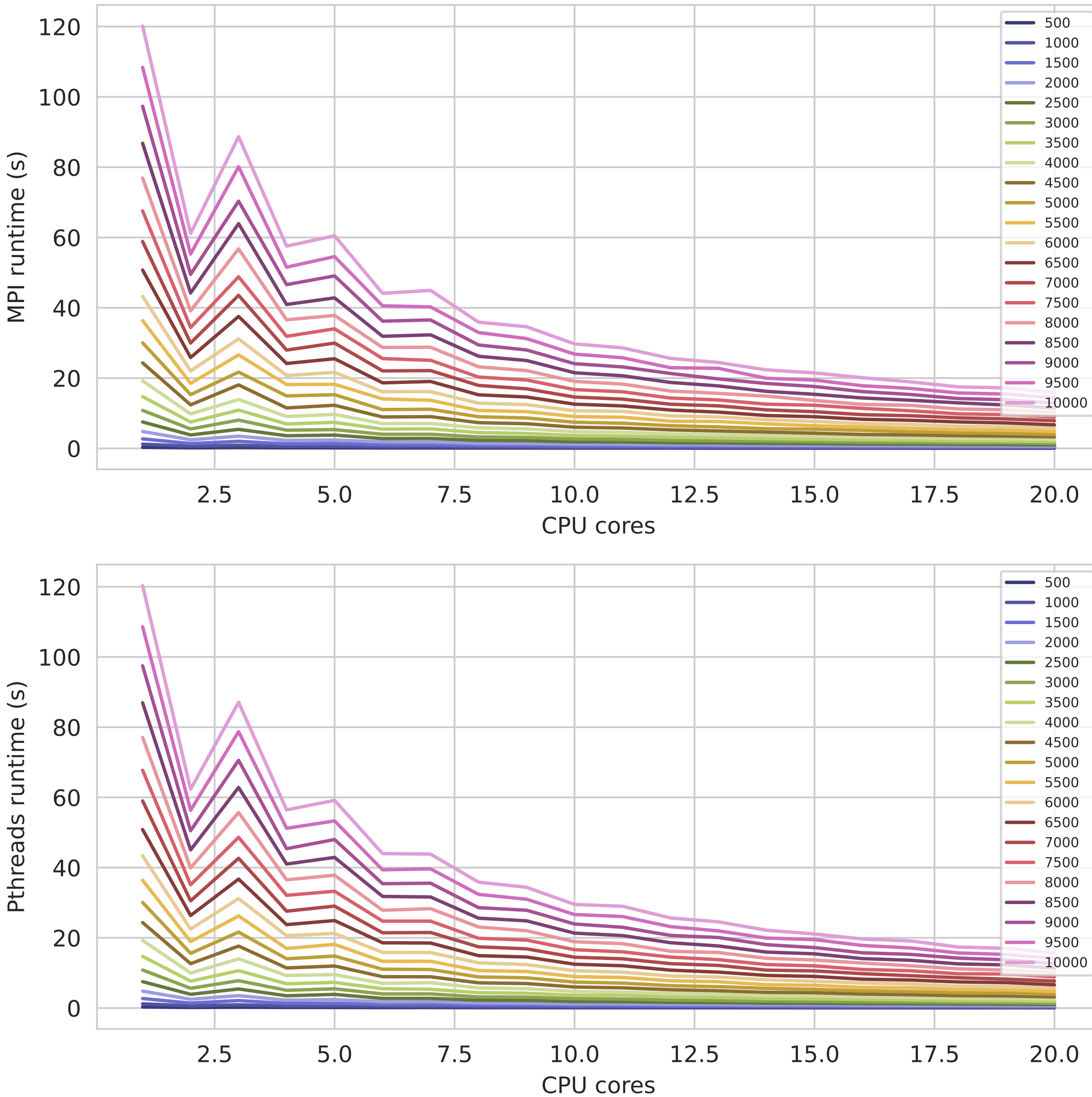


Figure 2: Runtime vs CPU cores plot.

## 3 Result and Discussion

### 3.1 Runtime

The graph of running time versus CPU cores and versus array size were plotted in Figure 2 and 3, respectively. From Figure 3, the plot clearly shows a perfect $O(n^2)$ the complexity of the algorithm, which is consistent with the theoretical analysis. From figure 2, however, for a fixed array size, the runtime does not monotonically decrease with the increase of CPU cores. It shows a zig-zag pattern from CPU cores ranging from 1 to 7. The reason behind this is the static scheduling

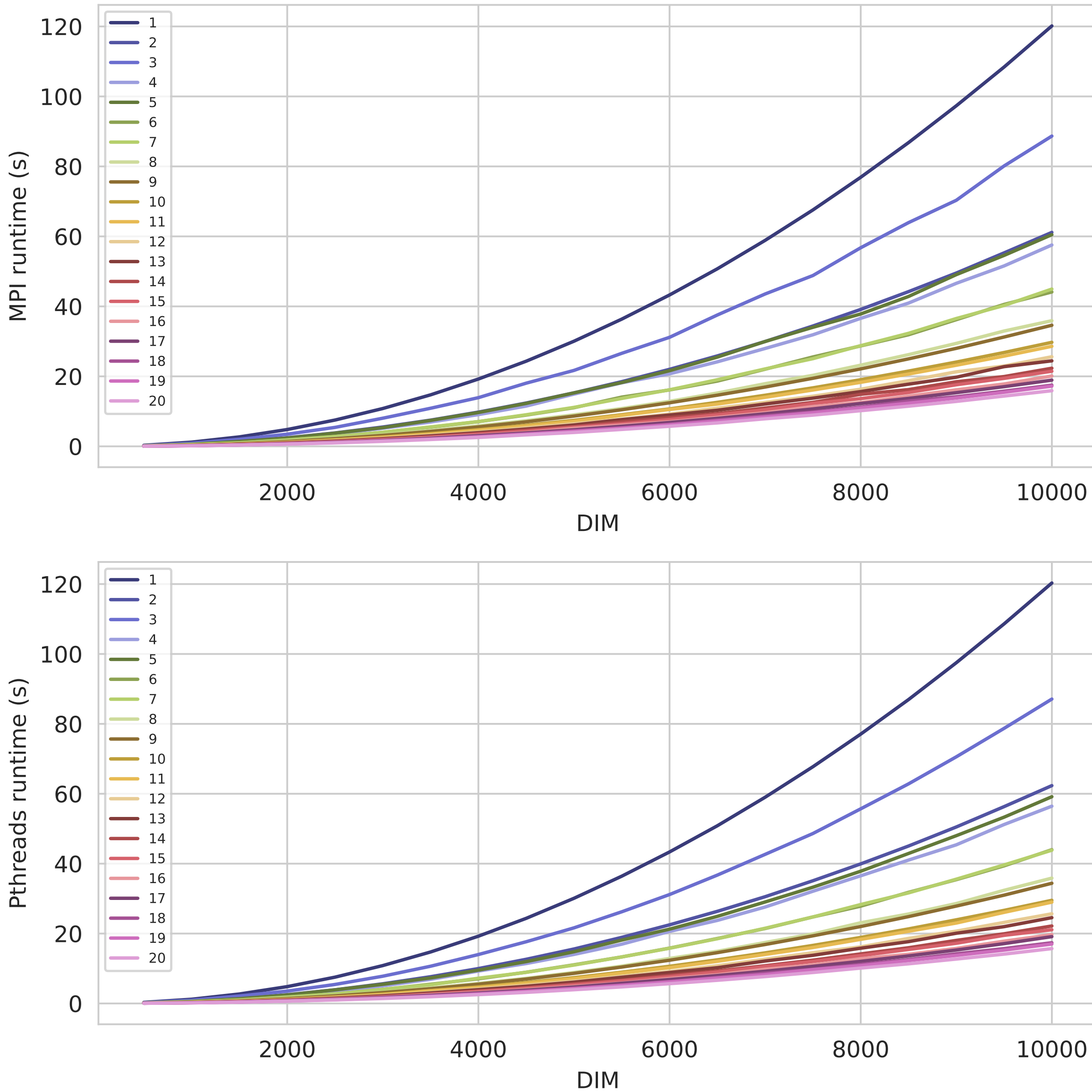


Figure 3: Runtime vs $x$-resolution plot.

algorithm used in the parallel program. According to the flowchart in Figure A.1 and A.2, the mesh is averagely distributed to each process/thread. Nevertheless, the computation time of each thread may not be the same, which means some thread may finish their job very fast and keep waiting other threads finish their job.

This issue would be further discussed in the following sections.

### 3.2 Performance analysis

The heatmap of acceleration is plotted in Figure 4. It should be noticed that when the array size is large ($n \geq 1000$), the speed-up ratio does not change a lot, which means the time spent on communication (MPI) and shared-memory operations (Pthreads) are comparatively a small part of the overall execution time. Moreover, from the first row of the heatmaps, the acceleration rate of Pthread is relatively lower than MPI's speed-up ratio, which indicates the creation and joining of threads could be time-consuming.

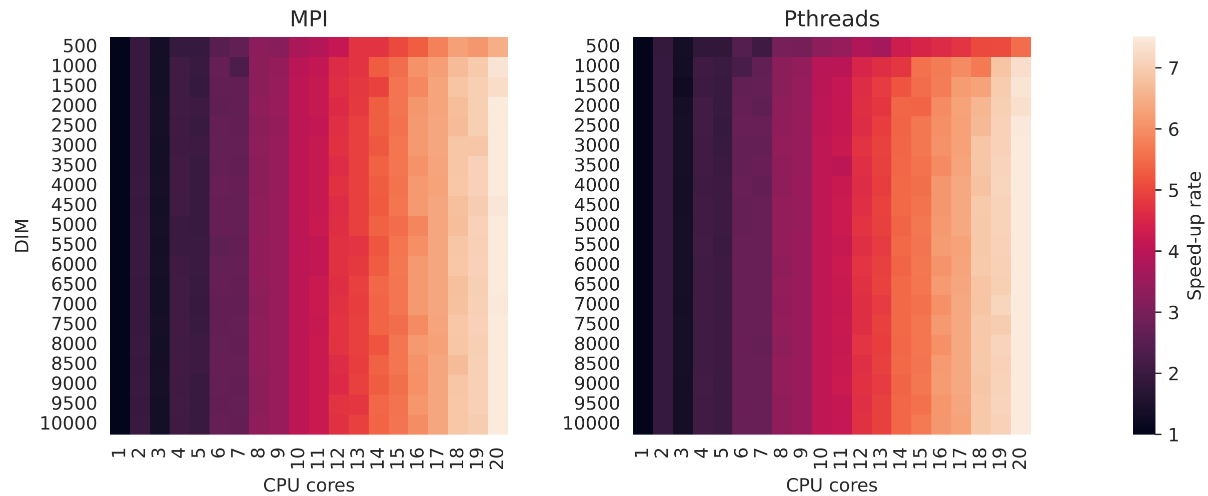


Figure 4: Speed-up ratio of two parallel programs.

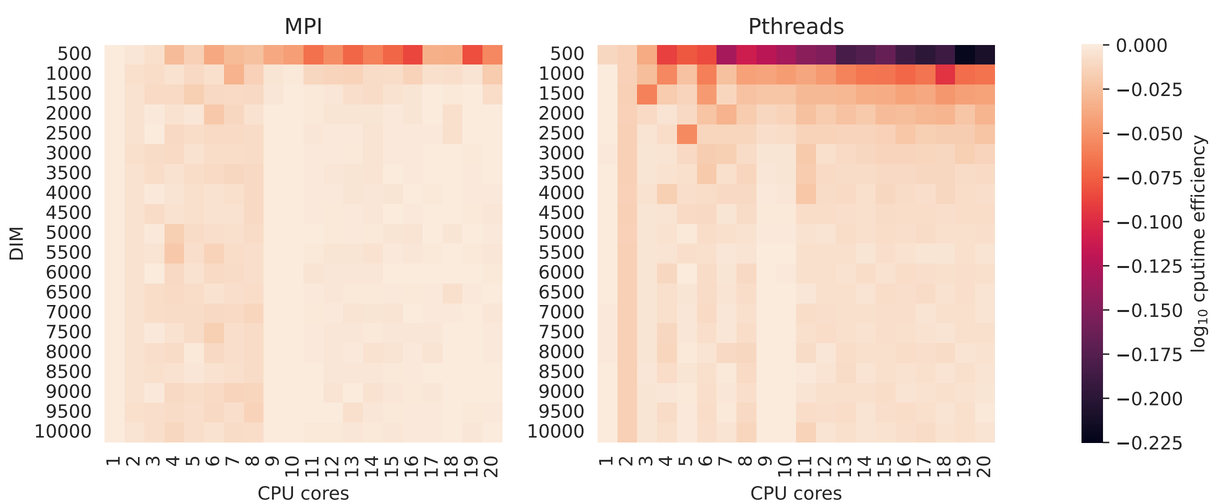


Figure 5: $\log_1 0$ CPU time efficiency.

The heatmap of CPU time spent on computation is plotted in Figure 5. It should be clear that when the array size is large, the computation efficiency of the parallel program is the same as the computational efficiency of the sequential program. Then, we can again confirm that the relative-low efficiency of the parallel program is caused by the scheduling algorithm which performs partition.

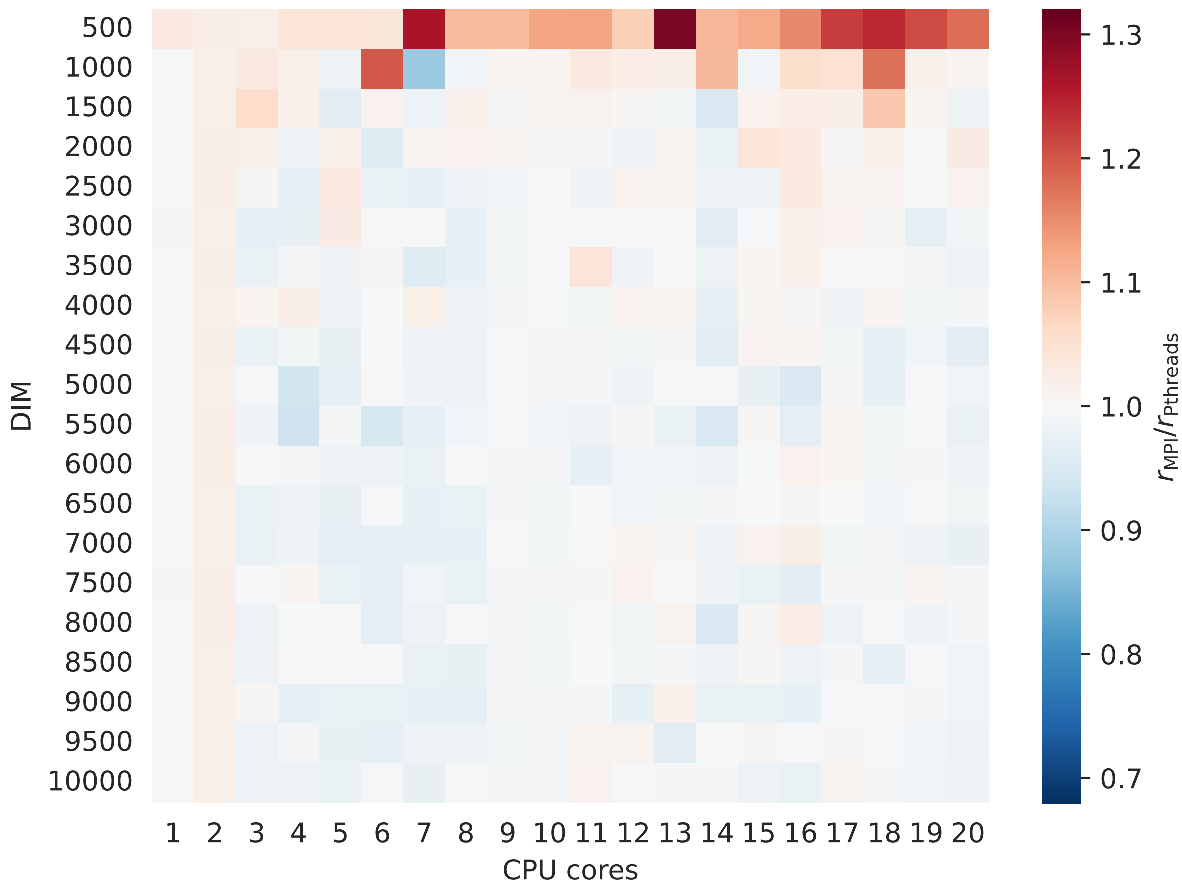


Figure 6: Ratio of MPI and Pthreads speed-up rate.

To solve this issue, one may apply dynamic scheduling. The dynamic scheduling would constantly check the status of each process/thread and utilize spare computational resources dynamically and efficiently.

### 3.3 Parallel schemes

Due to the static scheduling algorithm, the performance of MPI and Pthreads are approximately the same, as Figure A.1 shown.

## 4 Conclusion

In conclusion, two parallel computing schemes for Mandelbrot set were implemented and their performances were compared and evaluated. One should use dynamic scheduling when dealing with large-scale computations to fully utilize computational resources.

# A Supplementary figures

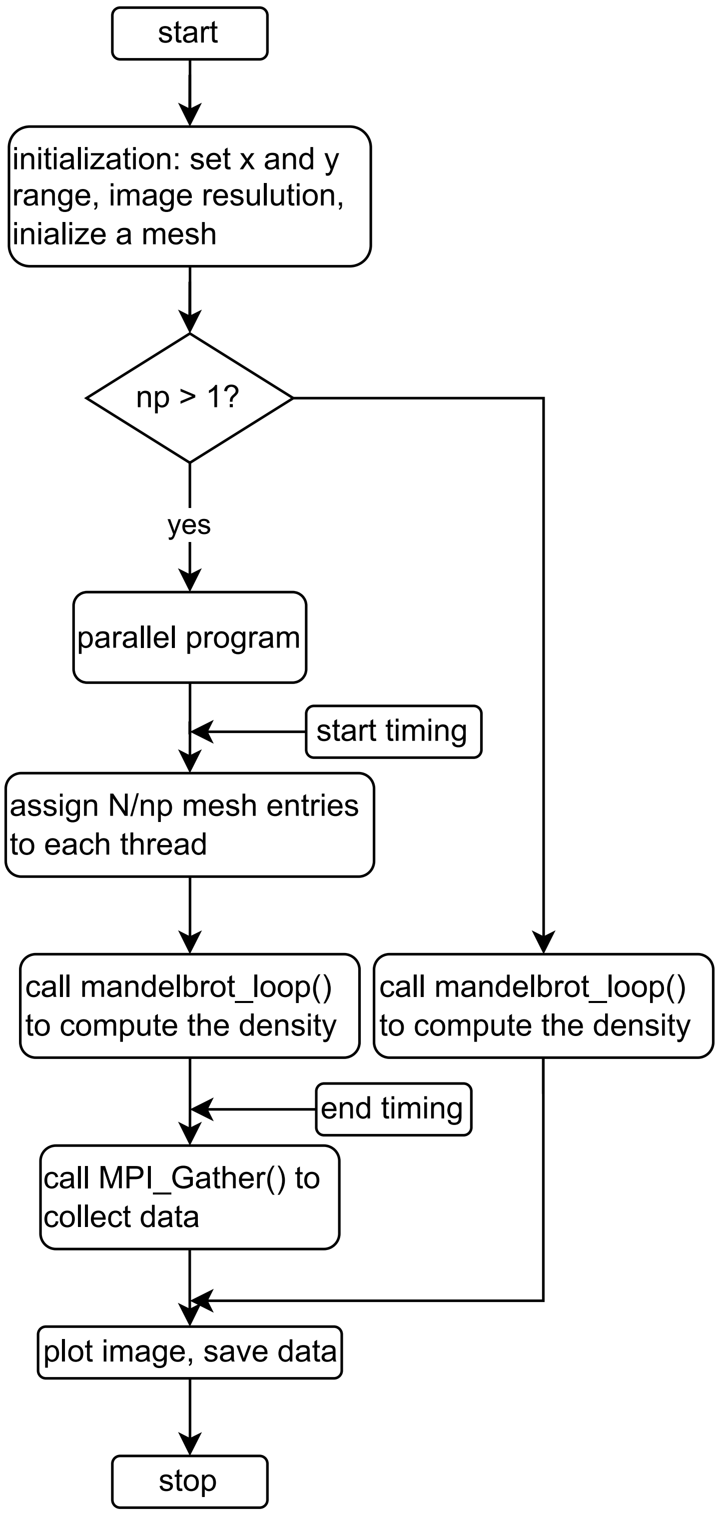


Figure A.1: MPI program flowchart

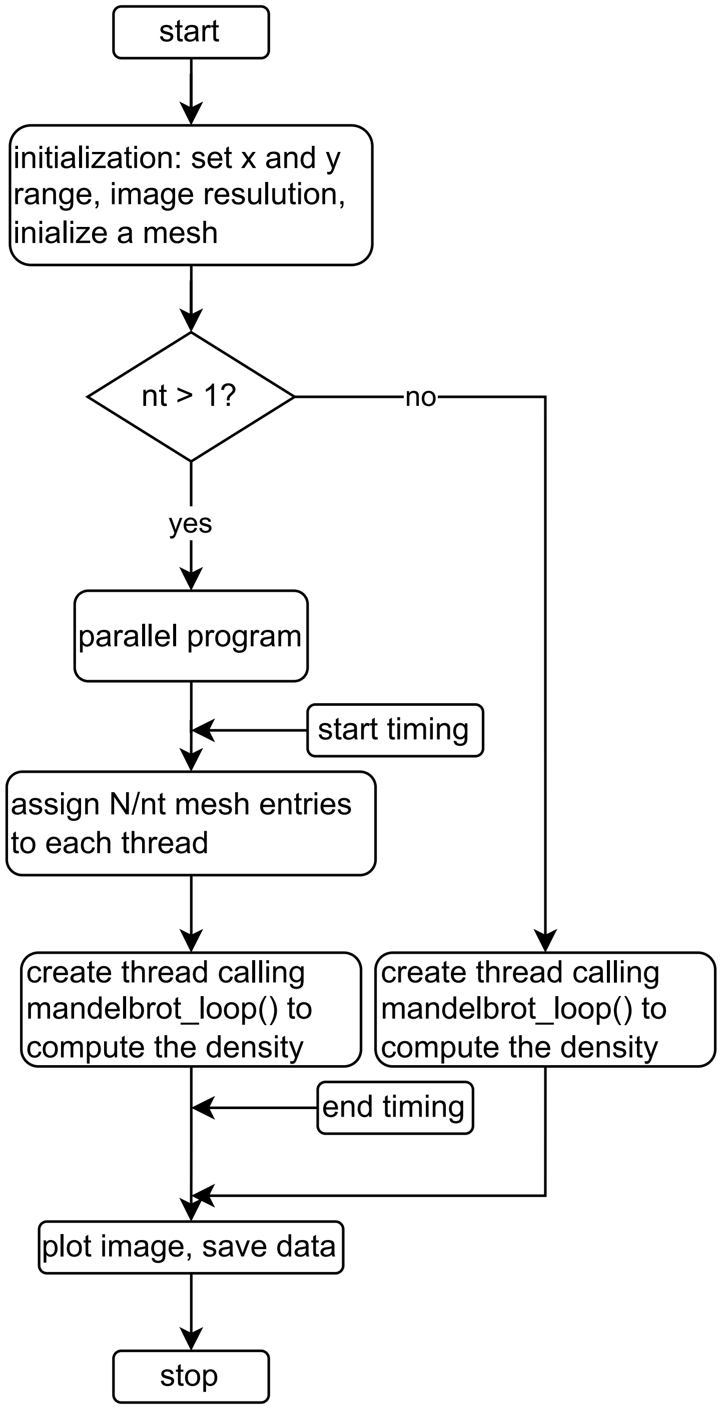


Figure A.2: Pthreads program flowchart

# B Source code

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.20)
project(hw01)

# set output path
set(CMAKE_LIBRARY_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/lib)
set(CMAKE_ARCHIVE_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/lib)
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/bin)

# set include libraires
include_directories(src)

set(CMAKE_CXX_STANDARD 11)

# add src folder
add_subdirectory(src)
```

src/CMakeLists.txt

```
find_package(MPI REQUIRED)

# options
# gui option
option(GUI "OPENGL Rendering" OFF)

# include & requirements
# pthread
set(THREADS_PREFER_PTHREAD_FLAG ON)
find_package(Threads REQUIRED)
# mpi
find_package(MPI REQUIRED)
include_directories(${MPI_INCLUDE_PATH})
# opengl & glut
if(GUI)
    find_package(OpenGL REQUIRED)
    find_package(GLUT REQUIRED)
    include_directories(${OPENGL_INCLUDE_DIRS} ${GLUT_DINCLUDE_DIRS})
    add_definitions(-DGUI)
endif()

# executable
add_executable(main.seq main.seq.cpp)
add_executable(main.pthread main.pthread.cpp)
add_executable(main.mpi main.mpi.cpp)
target_link_libraries(main.pthread Threads::Threads)
target_link_libraries(main.mpi ${MPI_LIBRARIES})
if(GUI)
    target_link_libraries(main.seq ${OPENGL_LIBRARIES} ${GLUT_LIBRARIES})
    target_link_libraries(main.mpi ${OPENGL_LIBRARIES} ${GLUT_LIBRARIES})
    target_link_libraries(main.pthread ${OPENGL_LIBRARIES} ${GLUT_LIBRARIES})
endif()
```

src/main.seq.cpp

```
#include <stdio.h>
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <string.h>
#include <chrono>
#include <thread>
#include "utils.h"
```

```
int main(int argc, char* argv[]) {
    // initialization
    float xmin = -2.0e-0;
    float xmax =  0.6e-0;
    float ymin = -1.3e-0;
    float ymax =  1.3e-0;
    int   DIM  =     500;
    int  save  =       1;
    int   iter =     200;
    int record =       0;

    // parse argument
    char buff[200];
    for (int i = 0; i < argc; i++){
        strcpy(buff, argv[i]);
        if (strcmp(buff, ”-n”)==0 || strcmp(buff, ”--ndim”)==0){
            std::string num(argv[i+1]);
            DIM = std::stoi(num);
        }
        if (strcmp(buff, ”--xmin”)==0){
            std::string num(argv[i+1]);
            xmin = std::stof(num);
        }
        if (strcmp(buff, ”--xmax”)==0){
            std::string num(argv[i+1]);
            xmax = std::stof(num);
        }
        if (strcmp(buff, ”--ymin”)==0){
            std::string num(argv[i+1]);
            ymin = std::stof(num);
        }
        if (strcmp(buff, ”--ymax”)==0){
            std::string num(argv[i+1]);
            ymax = std::stof(num);
        }
        if (strcmp(buff, ”--iter”)==0){
            std::string num(argv[i+1]);
            iter = std::stof(num);
        }
        if (strcmp(buff, ”--save”)==0){
            std::string num(argv[i+1]);
            save = std::stoi(num);
        }
        if (strcmp(buff, ”--record”)==0){
            std::string num(argv[i+1]);
            record = std::stoi(num);
        }
    }
    // postprocessing
    int xDIM = DIM;
    int yDIM = int(DIM*(ymax-ymin)/(xmax-xmin));

    // print info
    print_info(xDIM, yDIM);

    // allocation and initialization
    std::complex<float> *Z = (std::complex<float> *)malloc(sizeof(std::complex<float>)*yDIM*xDIM);
    char *map = (char *)malloc(sizeof(char) * xDIM * yDIM);
    mandelbrot_init(Z, xDIM, yDIM, xmin, xmax, ymin, ymax);

    // start time
    auto t1 = std::chrono::system_clock::now();

    // MAIN program
```

```
    mandelbrot_loop(Z, map, 0, xDIM*yDIM, iter);

    // end time
    auto t2 = std::chrono::system_clock::now();
    auto dur = t2 - t1;
    auto dur_ = std::chrono::duration_cast<std::chrono::duration<double>>(dur);
    double t = dur_.count();

    // record data
    if (record==1) runtime_record(”seq”, DIM, 1, t, t);

    // save png
    if (save==1) mandelbrot_save(”seq”, map, xDIM, yDIM);

    // end time
    runtime_print(DIM, 1, t, t);

    // rendering
    #ifdef GUI
    // copy memory
    map_glut = (char *)malloc(sizeof(char)*xDIM*yDIM);
    memcpy(map_glut, map, sizeof(char)*xDIM*yDIM);
    // plot
    xDIM_glut = xDIM;
    yDIM_glut = yDIM;
    render(”seq”);
    free(map_glut);
    #endif

    // free arrays
    free(Z);
    free(map);

    return 0;
}
```

src/main.mpi.cpp

```
#include <stdio.h>
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <string.h>
#include <chrono>
#include <thread>
#include <mpi.h>
#include ”utils.h”


int main(int argc, char* argv[]) {
    // mpi initializatio
    MPI_Init(NULL, NULL);
    // fetch size and rank
    int size, rank;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);


    // initialization
    float xmin = -2.0e-0;
    float xmax =  0.6e-0;
    float ymin = -1.3e-0;
    float ymax =  1.3e-0;
    int   DIM  =    500;
    int   save  =      1;
    int   iter =    200;
    int record =      0;
```

```

    // parse argument
    char buff[200];
    for (int i = 0; i < argc; i++){
        strcpy(buff, argv[i]);
        if (strcmp(buff, ”-n”)==0 || strcmp(buff, ”--ndim”)==0){
            std::string num(argv[i+1]);
            DIM = std::stoi(num);
        }
        if (strcmp(buff, ”--xmin”)==0){
            std::string num(argv[i+1]);
            xmin = std::stof(num);
        }
        if (strcmp(buff, ”--xmax”)==0){
            std::string num(argv[i+1]);
            xmax = std::stof(num);
        }
        if (strcmp(buff, ”--ymin”)==0){
            std::string num(argv[i+1]);
            ymin = std::stof(num);
        }
        if (strcmp(buff, ”--ymax”)==0){
            std::string num(argv[i+1]);
            ymax = std::stof(num);
        }
        if (strcmp(buff, ”--iter”)==0){
            std::string num(argv[i+1]);
            iter = std::stof(num);
        }
        if (strcmp(buff, ”--save”)==0){
            std::string num(argv[i+1]);
            save = std::stoi(num);
        }
        if (strcmp(buff, ”--record”)==0){
            std::string num(argv[i+1]);
            record = std::stoi(num);
        }
    }
    // postprocessing
    int xDIM = DIM;
    int yDIM = int(DIM*(ymax-ymin)/(xmax-xmin));

    // pre-defined variables
    std::complex<float> *Z;
    std::complex<float> *Z_;
    char *map;
    char *map_;
    int start_idx = xDIM*yDIM/size * rank;
    int end_idx = xDIM*yDIM/size * (rank+1);
    if (rank==size-1) end_idx = xDIM*yDIM;
    if (rank==0){
        // print info
        print_info(xDIM, yDIM);
        // allocation and initialization
        Z = (std::complex<float> *)malloc(sizeof(std::complex<float>)*yDIM*xDIM);
        map = (char *)malloc(sizeof(char) * xDIM * yDIM);
        mandelbrot_init(Z, xDIM, yDIM, xmin, xmax, ymin, ymax);
    }
    // allocate local variables for each process
    Z_ = (std::complex<float> *)malloc(sizeof(std::complex<float>) * (end_idx-start_idx));
    map_ = (char *)malloc(sizeof(char) * (end_idx-start_idx));

    // timing
    double t1, t2, t1_, t2_;

    // MAIN program
```

```
    // start timing
    t1 = MPI_Wtime();
    // CASE 1: sequential
    if (size==1){
        t1_ = MPI_Wtime();
        mandelbrot_loop(Z, map, 0, xDIM*yDIM, iter);
        t2_ = MPI_Wtime();
    }
    // CASE 2: parallel
    else {
        // distribute the data
        int scale = sizeof(std::complex<float>) / sizeof(int);
        if (rank==0) {
            for (int i = 1; i < size; i++){
                int start = xDIM*yDIM/size * i;
                int end   = xDIM*yDIM/size * (i+1); if (i==size-1) end = xDIM*yDIM;
                MPI_Send((int *) (Z+start), (end-start)*scale, MPI_INT, i, 0,
                    MPI_COMM_WORLD);
            }
            for (int i = 0; i < xDIM*yDIM/size; i++) Z_[i] = Z[i];
        }
        else {
            MPI_Recv((int *) Z_, (end_idx-start_idx)*scale, MPI_INT, 0, 0,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }
        // // print check
        // printf("rank %d start_idx %d end_idx %d print %f + %fi\n",
        //         rank, start_idx, end_idx, std::real(Z_[0]), std::imag(Z_[0]));

        // start timing
        t1_ = MPI_Wtime();

        // execution
        mandelbrot_loop(Z_, map_, 0, end_idx-start_idx, iter);

        // end timing
        t2_ = MPI_Wtime();

        // gather data
        MPI_Gather(map_, xDIM*yDIM/size, MPI_CHAR, map, xDIM*yDIM/size, MPI_CHAR, 0,
            MPI_COMM_WORLD);
        MPI_Barrier(MPI_COMM_WORLD);
        // tail case
        if (xDIM*yDIM%size != 0){
            if (rank == size-1) MPI_Send(map_+xDIM*yDIM/size, xDIM*yDIM%size, MPI_CHAR,
                0, 2, MPI_COMM_WORLD);
            if (rank == 0) MPI_Recv(map+xDIM*yDIM/size*size, xDIM*yDIM%size, MPI_CHAR,
                size-1, 2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }
        MPI_Barrier(MPI_COMM_WORLD);
    }
    // end timing
    t2 = MPI_Wtime();

    // end time
    double t = t2 - t1;    // overall execution time
    double t_ = t2_ - t1_; // cpu time on calculaiton
    double *time_arr = (double *)malloc(sizeof(double) * size);
    double t_sum = 0;
    MPI_Gather(&t_, 1, MPI_DOUBLE, time_arr, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    MPI_Barrier(MPI_COMM_WORLD);
    for (int i = 0; i < size; i++){
        t_sum += time_arr[i];
    }
    MPI_Barrier(MPI_COMM_WORLD);

    // record data
```

```
    if (rank==0 && record==1){
        runtime_record("mpi", DIM, size, t, t_sum);
        runtime_record_detail("mpi", DIM, size, t, time_arr);
    }

    // save png
    if (rank==0 && save==1) mandelbrot_save("mpi", map, xDIM, yDIM);

    // sync
    MPI_Barrier(MPI_COMM_WORLD);

    // end time
    if (rank==0) runtime_print(DIM, size, t, t_sum);

    // rendering
    #ifdef GUI
    if (rank==0){
        // copy memory
        map_glut = (char *)malloc(sizeof(char)*xDIM*yDIM);
        memcpy(map_glut, map, sizeof(char)*xDIM*yDIM);
        // plot
        xDIM_glut = xDIM;
        yDIM_glut = yDIM;
        render("seq");
        free(map_glut);
    }
    #endif

    // free arrays
    if (rank==0){
        free(Z);
        free(map);
    }
    free(Z_);
    free(map_);
    MPI_Barrier(MPI_COMM_WORLD);

    // mpi finalization
    MPI_Finalize();

    return 0;
}
```

src/main.pthread.cpp

```
#include <stdio.h>
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <string.h>
#include <chrono>
#include <thread>
#include <pthread.h>
#include "utils.h"


int main(int argc, char* argv[]) {
    // initialization
    float xmin = -2.0e-0;
    float xmax =  0.6e-0;
    float ymin = -1.3e-0;
    float ymax =  1.3e-0;
    int   DIM  =    500;
    int   save  =      1;
    int   iter =    200;
    int record =      0;

```

```
    // pthread specific args
    int     nt =      1;

    // parse argument
    char buff[200];
    for (int i = 0; i < argc; i++){
        strcpy(buff, argv[i]);
        if (strcmp(buff, "-n")==0 || strcmp(buff, "--ndim")==0){
            std::string num(argv[i+1]);
            DIM = std::stoi(num);
        }
        if (strcmp(buff, "-nt")==0 || strcmp(buff, "--nthread")==0){
            std::string num(argv[i+1]);
            nt = std::stoi(num);
        }
        if (strcmp(buff, "--xmin")==0){
            std::string num(argv[i+1]);
            xmin = std::stof(num);
        }
        if (strcmp(buff, "--xmax")==0){
            std::string num(argv[i+1]);
            xmax = std::stof(num);
        }
        if (strcmp(buff, "--ymin")==0){
            std::string num(argv[i+1]);
            ymin = std::stof(num);
        }
        if (strcmp(buff, "--ymax")==0){
            std::string num(argv[i+1]);
            ymax = std::stof(num);
        }
        if (strcmp(buff, "--iter")==0){
            std::string num(argv[i+1]);
            iter = std::stof(num);
        }
        if (strcmp(buff, "--save")==0){
            std::string num(argv[i+1]);
            save = std::stoi(num);
        }
        if (strcmp(buff, "--record")==0){
            std::string num(argv[i+1]);
            record = std::stoi(num);
        }
    }
    // postprocessing
    int xDIM = DIM;
    int yDIM = int(DIM*(ymax-ymin)/(xmax-xmin));

    // print info
    print_info(xDIM, yDIM);

    // allocation and initialization
    std::complex<float> *Z = (std::complex<float> *)malloc(sizeof(std::complex<float>)*
        yDIM*xDIM);
    char *map = (char *)malloc(sizeof(char) * xDIM * yDIM);
    mandelbrot_init(Z, xDIM, yDIM, xmin, xmax, ymin, ymax);
    Ptargs *args = (Ptargs *)malloc(sizeof(Ptargs) * nt);
    pthread_t *threads = (pthread_t *)malloc(sizeof(pthread_t) * nt);

    // start time
    auto t1 = std::chrono::system_clock::now();
    double *time_arr = (double *)malloc(sizeof(double)*nt);

    // MAIN program
    // create threads
    for (int i = 0; i < nt; i++){
        // calculate start and end index
```

```
        int start_idx = xDIM*yDIM/nt * i;
        int end_idx = xDIM*yDIM/nt * (i+1);
        args[i] = (Ptargs){.Z=Z, .map=map, .start_idx=start_idx, .end_idx=end_idx, .iter
            =iter, .id=i, .time_arr=time_arr};
        if (i==nt-1) args[i].end_idx = xDIM*yDIM;

        // create independent threads
        pthread_create(&threads[i], NULL, mandelbrot_loop_pt, (void *)(&args[i]));
    }
    // join threads
    for (int i = 0; i < nt; i++){
        pthread_join(threads[i], NULL);
    }

    // end time
    auto t2 = std::chrono::system_clock::now();
    auto dur = t2 - t1;
    auto dur_ = std::chrono::duration_cast<std::chrono::duration<double>>(dur);
    double t = dur_.count();
    double t_sum = 0;
    for (int i = 0; i < nt; i++) t_sum += time_arr[i];

    // record data
    if (record==1){
        runtime_record("pth", DIM, nt, t, t_sum);
        runtime_record_detail("pth", DIM, nt, t, time_arr);
    }

    // save png
    if (save==1) mandelbrot_save("pth", map, xDIM, yDIM);

    // end time
    runtime_print(DIM, nt, t, t_sum);

    // rendering
    #ifdef GUI
    // copy memory
    map_glut = (char *)malloc(sizeof(char)*xDIM*yDIM);
    memcpy(map_glut, map, sizeof(char)*xDIM*yDIM);
    // plot
    xDIM_glut = xDIM;
    yDIM_glut = yDIM;
    render("seq");
    free(map_glut);
    #endif


    // free arrays
    free(Z);
    free(map);

    return 0;
}
```

src/utils.h

```
#define STB_IMAGE_WRITE_IMPLEMENTATION
#include <stdio.h>
#include <iostream>
#include <complex>
#include "stb_image_write.h"
#include <sys/stat.h>
#include <sys/types.h>

#ifdef GUI
#include <GL/glut.h>
#include <GL/glu.h>
```

```
#include <GL/gl.h>

char *map_glut;
int xDIM_glut, yDIM_glut;
int width = 1000;
int xwidth, ywidth;
#endif

typedef struct ptargs{
    std::complex<float> *Z;
    char *map;
    int start_idx;
    int end_idx;
    int iter;
    int id;
    double *time_arr;
} Ptargs;

void print_info(int xDIM, int yDIM){
    printf("Name: Haoran Sun\n");
    printf("ID:   119010271\n");
    printf("HW:   Mandelbrot Set Computation\n");
    printf("Set xDIM to %d, yDIM to %d\n", xDIM, yDIM);
}

void mandelbrot_init(std::complex<float> *Z, int xDIM, int yDIM, float xmin, float xmax,
     float ymin, float ymax){
    for (int i = 0; i < yDIM; i++){
        for (int j = 0; j < xDIM; j++){
            float x = (xmax-xmin)/xDIM*j + xmin;
            float y = (ymin-ymax)/yDIM*i + ymax;
            // printf("%f %f\n", x, y);
            Z[i*xDIM+j] = std::complex<float>(x, y);
        }
    }
}

char mandelbrot_iter(std::complex<float> z, std::complex<float> z0, int iter){
    std::complex<float> p = z;
    for (int i = 0; i < iter; i++){
        z = z * z + z0;
        if (std::real(z * std::conj(z)) > 4) return 255 - 255 * i/iter;
    }
    return 0;
}

void mandelbrot_loop(std::complex<float> *Z, char *map, int start_idx, int end_idx, int
    iter){
    for (int i = start_idx; i < end_idx; i++){
        map[i] = mandelbrot_iter(Z[i], Z[i], iter);
    }
}

void *mandelbrot_loop_pt(void *vargs){
    // transfer args
    Ptargs args = *(Ptargs *)vargs;
    double *time_arr = args.time_arr;
    int id = args.id;
    // start time
    auto t1 = std::chrono::system_clock::now();

    // main loop
    mandelbrot_loop(args.Z, args.map, args.start_idx, args.end_idx, args.iter);

    // end time
    auto t2 = std::chrono::system_clock::now();
    auto dur = t2 - t1;
```

```
    auto dur_ = std::chrono::duration_cast<std::chrono::duration<double>>(dur);
    double t = dur_.count();
    time_arr[id] =  t;

    return NULL;
}

void mandelbrot_save(const char *jobtype, char *map,
    int xDIM, int yDIM){
    char filebuff[200];
    snprintf(filebuff, sizeof(filebuff), "mandelbrot_%s.png", jobtype);
    stbi_write_png(filebuff, xDIM, yDIM, 1, map, 0);
    printf("Image saved as %s.\n", filebuff);
}

void runtime_record(const char *jobtype, int N, int nt, double t, double t_sum){
    const char *folder = "data";
    mkdir(folder, 0777);
    FILE* outfile;
    char filebuff[200];
    snprintf(filebuff, sizeof(filebuff), "./%s/runtime_%s.txt", folder, jobtype);
    outfile = fopen(filebuff, "a");
    fprintf(outfile, "%10d %5d %10.4f %10.4f\n", N, nt, t, t_sum);
    fclose(outfile);
    printf("Runtime added in %s.\n", filebuff);
}

void runtime_record_detail(const char *jobtype, int N, int nt, double t, double *
    time_arr){
    const char *folder = "data";
    mkdir(folder, 0777);
    FILE* outfile;
    char filebuff[200];
    snprintf(filebuff, sizeof(filebuff), "./%s/runtime_detailed_%s_%d.txt", folder,
        jobtype, nt);
    outfile = fopen(filebuff, "a");
    fprintf(outfile, "%10d %5d %10.4f ", N, nt, t);
    for (int i = 0; i < nt; i++){
        fprintf(outfile, "%10.4f ", time_arr[i]);
    }
    fprintf(outfile, "\n");
    fclose(outfile);
    printf("Detailed runtime added in %s.\n", filebuff);
}

void runtime_print(int N, int nt, double t, double t_sum){
    printf("Execution time: %.2fs, cpu time: %.2fs, #cpu %2d\n", t, t_sum, nt);
}

#ifdef GUI

void display_test(){
    glClear(GL_COLOR_BUFFER_BIT);

    glBegin(GL_POLYGON);
    glVertex2f(0, 0);
    glVertex2f(1, 0);
    glVertex2f(1, 1);
    glVertex2f(0, 1);
    glEnd();

    glFlush();
}

void plot(){
    // display test
    // initialization
```

```
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(0.0f, 0.0f, 0.0f);

    // draw points
    GLfloat pointSize = 1.0f;
    glPointSize(pointSize);
    glBegin(GL_POINTS);
        glClear(GL_COLOR_BUFFER_BIT);
        for (int i = 0; i < yDIM_glut; i++){
            for (int j = 0; j < xDIM_glut; j++){
                int c0 = (unsigned char) map_glut[i*xDIM_glut+j];
                float c = c0;
                c = c0 / 255.0;
                glColor3f(c, c, c);
                glVertex2f(j, yDIM_glut-i);
            }
        }
    glEnd();

    // flush
    glFlush();
}

void resize(int x, int y){
    glutReshapeWindow(xwidth, ywidth);
}


void render(const char *jobtype){
    // glu init
    int glufoo = 1;
    char q[] = " ";
    char *glubar[1];
    glubar[0] = q;
    glutInit(&glufoo, glubar);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);

    // set x and y width
    xwidth = width;
    ywidth = yDIM_glut*width/xDIM_glut;
    glutInitWindowSize(xwidth, ywidth);
    glutCreateWindow(jobtype);
    glMatrixMode(GL_PROJECTION);
        gluOrtho2D(0, xDIM_glut, 0, yDIM_glut);

    // display func
    glutDisplayFunc(plot);
    // glutDisplayFunc(display_test);
    glutReshapeFunc(resize);

    glutMainLoop();
}

#endif
```