

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 $z_0 = c$.

$$z_{n+1}^2 = z_n^2 + c \quad (1)$$

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`.

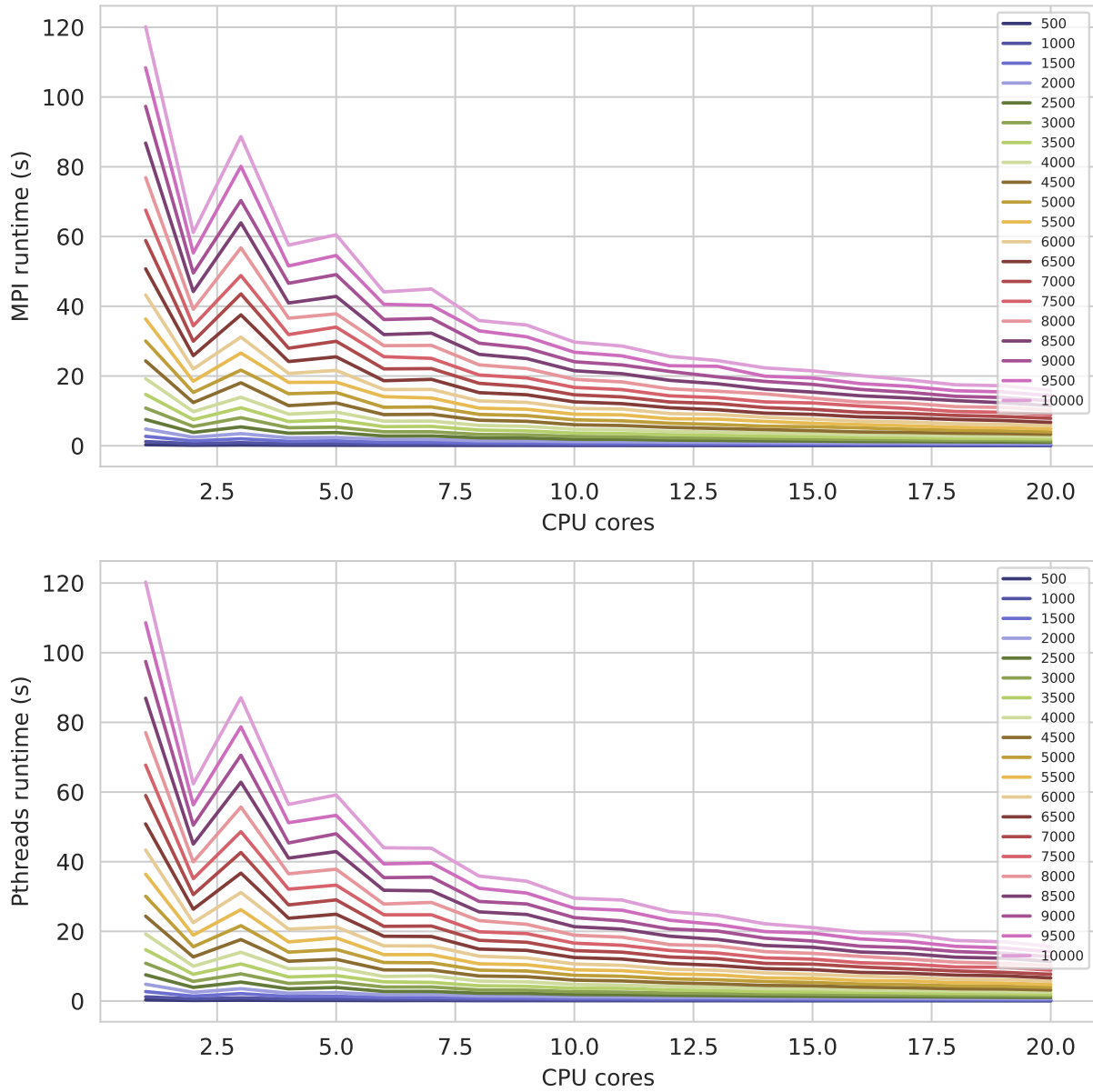


Figure 2: Runtime vs CPU cores plot.

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

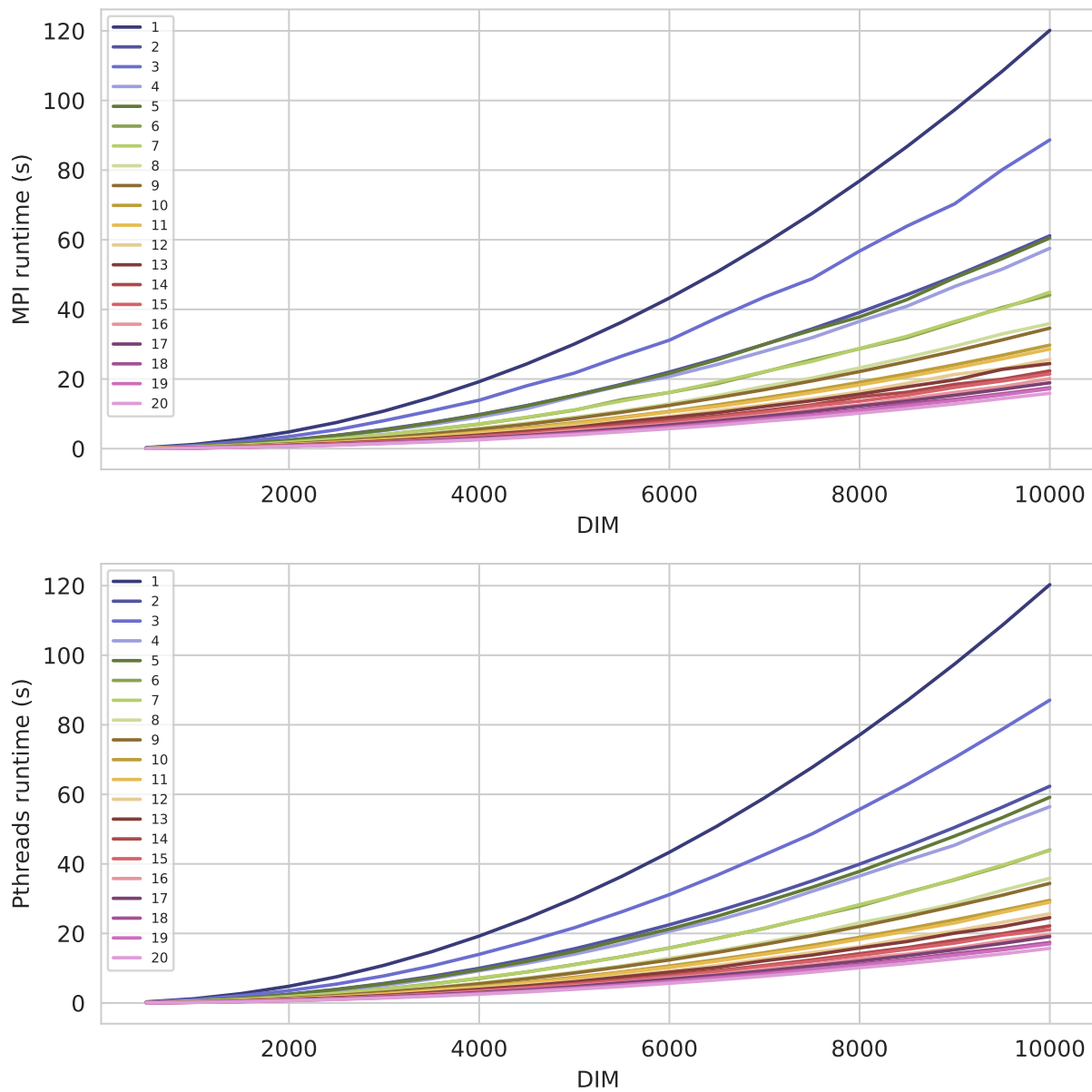


Figure 3: Runtime vs x -resolution plot.

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.

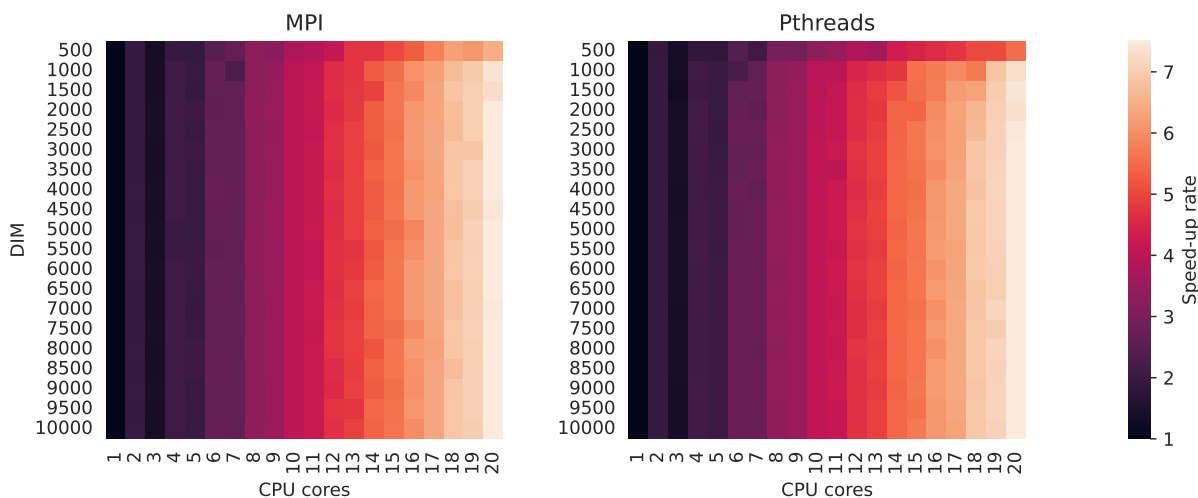


Figure 4: Speed-up ratio of two parallel programs.

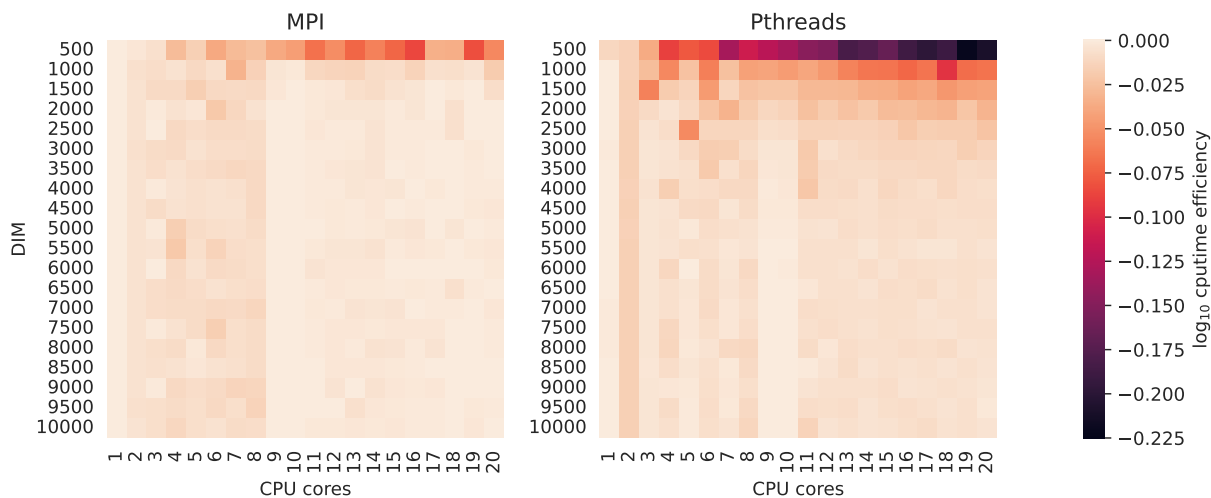


Figure 5: \log_{10} CPU time efficiency.

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.

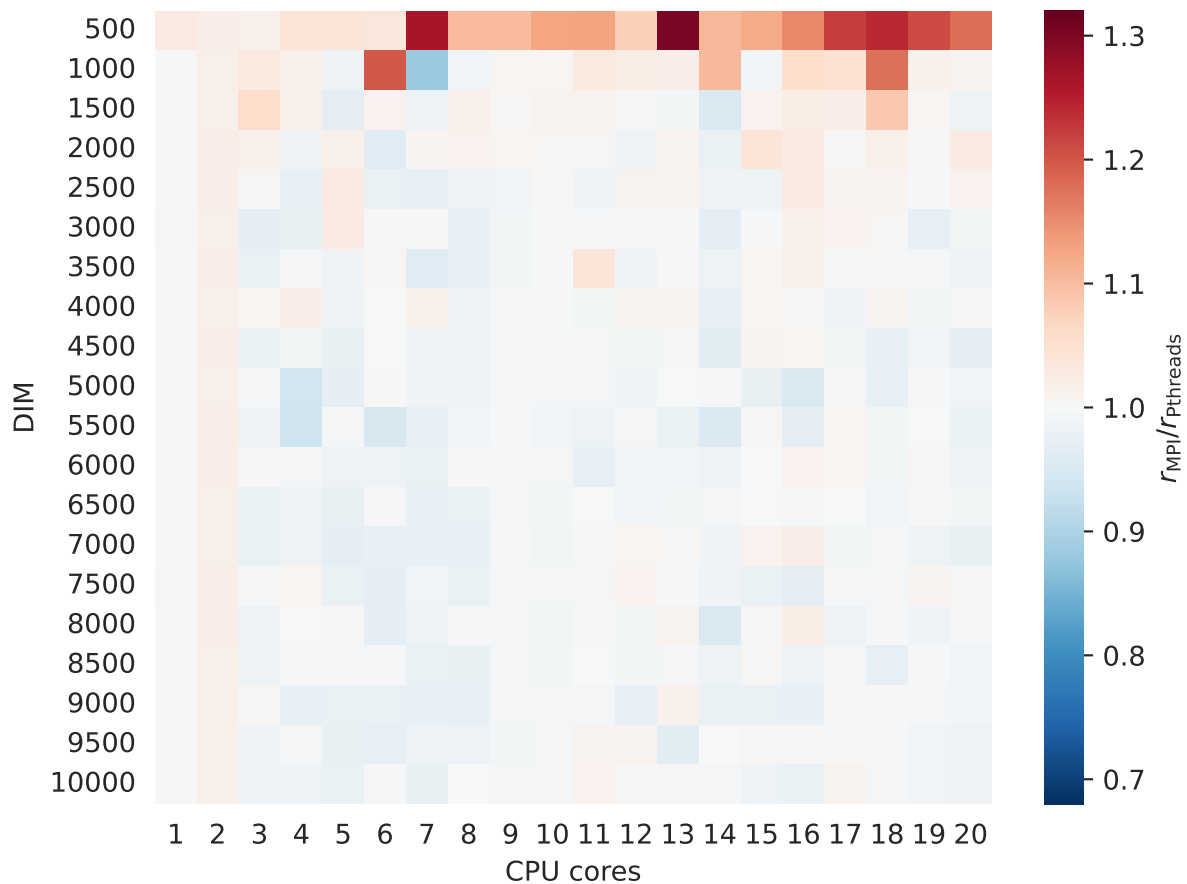


Figure 6: Ratio of MPI and Pthreads speed-up rate.

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

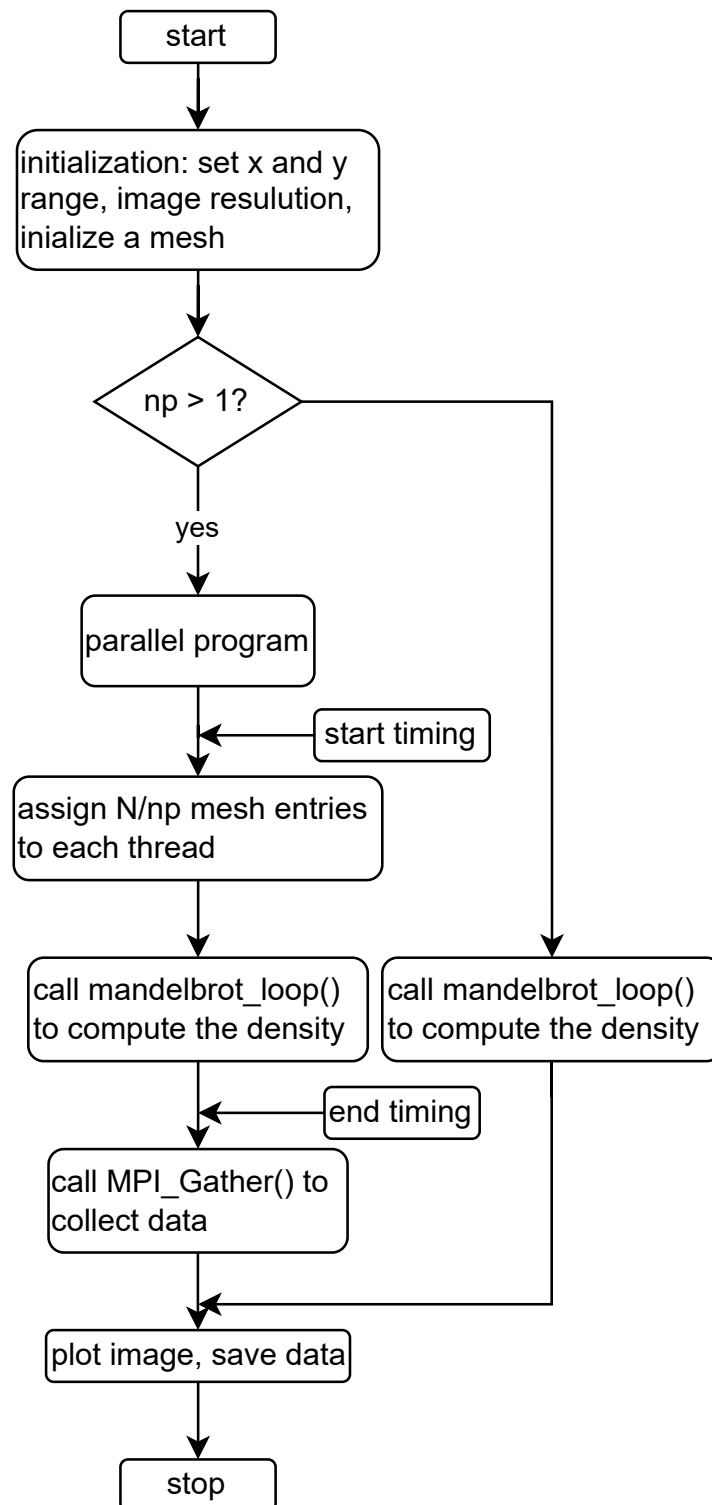


Figure A.1: MPI program flowchart

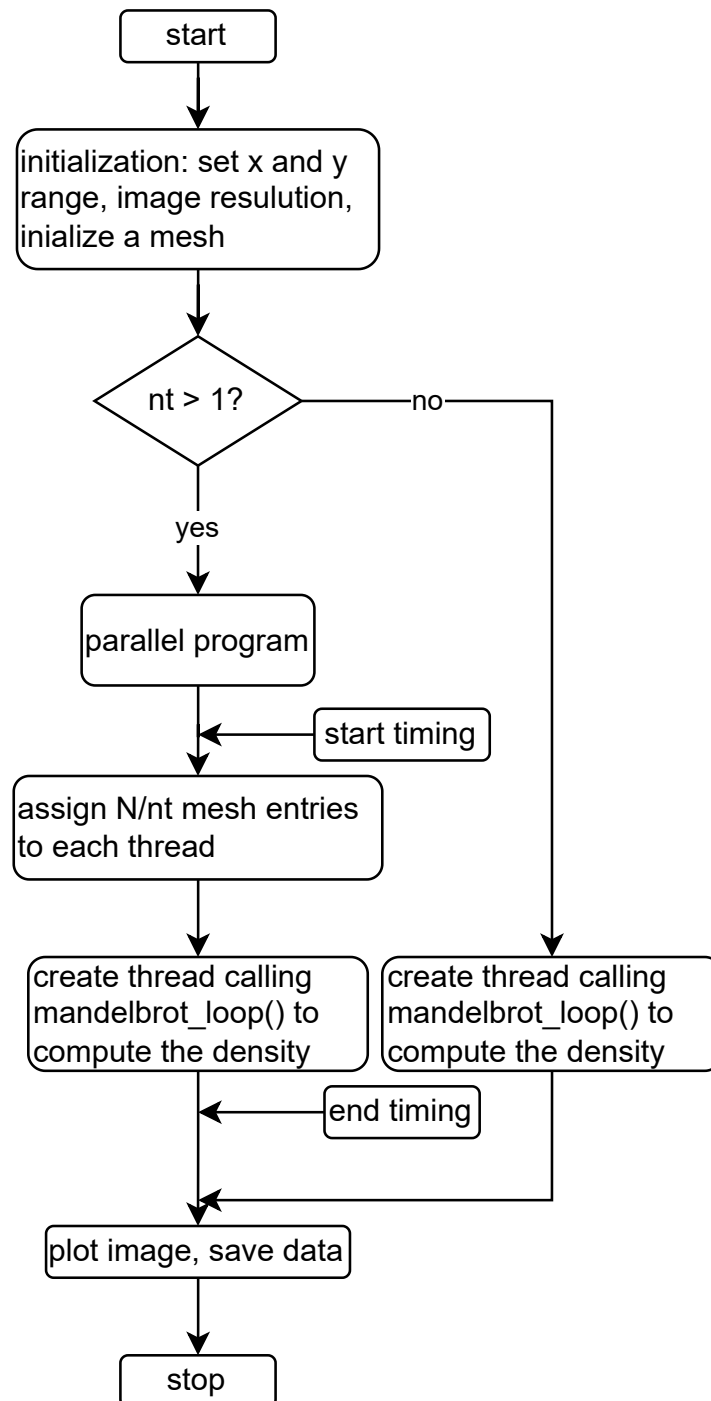


Figure A.2: Pthreads program flowchart

B Source code

CMakeLists.txt

```

1 cmake_minimum_required(VERSION 3.20)
2 project(hw01)
3
4 # set output path
5 set(CMAKE_LIBRARY_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/lib)
6 set(CMAKE_ARCHIVE_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/lib)
7 set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/bin)
8
9 # set include libraires
10 include_directories(src)
11
12 set(CMAKE_CXX_STANDARD 11)
13
14 # add src folder
15 add_subdirectory(src)

```

src/CMakeLists.txt

```

1 find_package(MPI REQUIRED)
2
3 # options
4 # gui option
5 option(GUI "OPENGL Rendering" OFF)
6
7 # include & requirements
8 # pthread
9 set(THREADS_PREFER_PTHREAD_FLAG ON)
10 find_package(Threads REQUIRED)
11 # mpi
12 find_package(MPI REQUIRED)
13 include_directories(${MPI_INCLUDE_PATH})
14 # opengl & glut
15 if(GUI)
16     find_package(OpenGL REQUIRED)
17     find_package(GLUT REQUIRED)
18     include_directories(${OPENGL_INCLUDE_DIRS} ${GLUT_INCLUDE_DIRS})
19     add_definitions(-DGUI)
20 endif()
21
22 # executable
23 add_executable(main.seq main.seq.cpp)
24 add_executable(main.pthread main.pthread.cpp)
25 add_executable(main.mpi main.mpi.cpp)
26 target_link_libraries(main.pthread Threads::Threads)
27 target_link_libraries(main.mpi ${MPI_LIBRARIES})
28 if(GUI)
29     target_link_libraries(main.seq ${OPENGL_LIBRARIES} ${GLUT_LIBRARIES})
30     target_link_libraries(main.mpi ${OPENGL_LIBRARIES} ${GLUT_LIBRARIES})
31     target_link_libraries(main.pthread ${OPENGL_LIBRARIES} ${GLUT_LIBRARIES})
32 endif()

```

src/main.seq.cpp

```

1 #include <stdio.h>
2 #include <iostream>
3 #include <fstream>
4 #include <cstdlib>
5 #include <string.h>
6 #include <chrono>
7 #include <thread>
8 #include "utils.h"

```

```

9
10
11 int main(int argc, char* argv[]) {
12     // initialization
13     float xmin = -2.0e-0;
14     float xmax = 0.6e-0;
15     float ymin = -1.3e-0;
16     float ymax = 1.3e-0;
17     int DIM = 500;
18     int save = 1;
19     int iter = 200;
20     int record = 0;
21
22     // parse argument
23     char buff[200];
24     for (int i = 0; i < argc; i++){
25         strcpy(buff, argv[i]);
26         if (strcmp(buff, "-n")==0 || strcmp(buff, "--ndim")==0){
27             std::string num(argv[i+1]);
28             DIM = std::stoi(num);
29         }
30         if (strcmp(buff, "--xmin")==0){
31             std::string num(argv[i+1]);
32             xmin = std::stof(num);
33         }
34         if (strcmp(buff, "--xmax")==0){
35             std::string num(argv[i+1]);
36             xmax = std::stof(num);
37         }
38         if (strcmp(buff, "--ymin")==0){
39             std::string num(argv[i+1]);
40             ymin = std::stof(num);
41         }
42         if (strcmp(buff, "--ymax")==0){
43             std::string num(argv[i+1]);
44             ymax = std::stof(num);
45         }
46         if (strcmp(buff, "--iter")==0){
47             std::string num(argv[i+1]);
48             iter = std::stof(num);
49         }
50         if (strcmp(buff, "--save")==0){
51             std::string num(argv[i+1]);
52             save = std::stoi(num);
53         }
54         if (strcmp(buff, "--record")==0){
55             std::string num(argv[i+1]);
56             record = std::stoi(num);
57         }
58     }
59     // postprocessing
60     int xDIM = DIM;
61     int yDIM = int(DIM*(ymax-ymin)/(xmax-xmin));
62
63     // print info
64     print_info(xDIM, yDIM);
65
66     // allocation and initialization
67     std::complex<float> *Z = (std::complex<float> *)malloc(sizeof(std::complex<float>)*
        yDIM*xDIM);
68     char *map = (char *)malloc(sizeof(char) * xDIM * yDIM);
69     mandelbrot_init(Z, xDIM, yDIM, xmin, xmax, ymin, ymax);
70
71     // start time
72     auto t1 = std::chrono::system_clock::now();
73
74     // MAIN program

```

```

75     mandelbrot_loop(Z, map, 0, xDIM*yDIM, iter);
76
77     // end time
78     auto t2 = std::chrono::system_clock::now();
79     auto dur = t2 - t1;
80     auto dur_ = std::chrono::duration_cast<std::chrono::duration<double>>(dur);
81     double t = dur_.count();
82
83     // record data
84     if (record==1) runtime_record("seq", DIM, 1, t, t);
85
86     // save png
87     if (save==1) mandelbrot_save("seq", map, xDIM, yDIM);
88
89     // end time
90     runtime_print(DIM, 1, t, t);
91
92     // rendering
93     #ifdef GUI
94     // copy memory
95     map_glut = (char *)malloc(sizeof(char)*xDIM*yDIM);
96     memcpy(map_glut, map, sizeof(char)*xDIM*yDIM);
97     // plot
98     xDIM_glut = xDIM;
99     yDIM_glut = yDIM;
100    render("seq");
101    free(map_glut);
102    #endif
103
104    // free arrays
105    free(Z);
106    free(map);
107
108    return 0;
109 }

```

src/main.mpi.cpp

```

1  #include <stdio.h>
2  #include <iostream>
3  #include <fstream>
4  #include <cstdlib>
5  #include <string.h>
6  #include <chrono>
7  #include <thread>
8  #include <mpi.h>
9  #include "utils.h"
10
11
12  int main(int argc, char* argv[]) {
13      // mpi initializatio
14      MPI_Init(NULL, NULL);
15      // fetch size and rank
16      int size, rank;
17      MPI_Comm_size(MPI_COMM_WORLD, &size);
18      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
19
20
21      // initialization
22      float xmin = -2.0e-0;
23      float xmax = 0.6e-0;
24      float ymin = -1.3e-0;
25      float ymax = 1.3e-0;
26      int DIM = 500;
27      int save = 1;
28      int iter = 200;
29      int record = 0;

```

```

30
31 // parse argument
32 char buff[200];
33 for (int i = 0; i < argc; i++){
34     strcpy(buff, argv[i]);
35     if (strcmp(buff, "-n")==0 || strcmp(buff, "--ndim")==0){
36         std::string num(argv[i+1]);
37         DIM = std::stoi(num);
38     }
39     if (strcmp(buff, "--xmin")==0){
40         std::string num(argv[i+1]);
41         xmin = std::stof(num);
42     }
43     if (strcmp(buff, "--xmax")==0){
44         std::string num(argv[i+1]);
45         xmax = std::stof(num);
46     }
47     if (strcmp(buff, "--ymin")==0){
48         std::string num(argv[i+1]);
49         ymin = std::stof(num);
50     }
51     if (strcmp(buff, "--ymax")==0){
52         std::string num(argv[i+1]);
53         ymax = std::stof(num);
54     }
55     if (strcmp(buff, "--iter")==0){
56         std::string num(argv[i+1]);
57         iter = std::stof(num);
58     }
59     if (strcmp(buff, "--save")==0){
60         std::string num(argv[i+1]);
61         save = std::stoi(num);
62     }
63     if (strcmp(buff, "--record")==0){
64         std::string num(argv[i+1]);
65         record = std::stoi(num);
66     }
67 }
68 // postprocessing
69 int xDIM = DIM;
70 int yDIM = int(DIM*(ymax-ymin)/(xmax-xmin));
71
72 // pre-defined variables
73 std::complex<float> *Z;
74 std::complex<float> *Z_;
75 char *map;
76 char *map_;
77 int start_idx = xDIM*yDIM/size * rank;
78 int end_idx = xDIM*yDIM/size * (rank+1);
79 if (rank==size-1) end_idx = xDIM*yDIM;
80 if (rank==0){
81     // print info
82     print_info(xDIM, yDIM);
83     // allocation and initialization
84     Z = (std::complex<float> *)malloc(sizeof(std::complex<float>)*yDIM*xDIM);
85     map = (char *)malloc(sizeof(char) * xDIM * yDIM);
86     mandelbrot_init(Z, xDIM, yDIM, xmin, xmax, ymin, ymax);
87 }
88 // allocate local variables for each process
89 Z_ = (std::complex<float> *)malloc(sizeof(std::complex<float>) * (end_idx-start_idx)
90 );
91 map_ = (char *)malloc(sizeof(char) * (end_idx-start_idx));
92
93 // timing
94 double t1, t2, t1_, t2_;
95
96 // MAIN program

```

```

96 // start timing
97 t1 = MPI_Wtime();
98 // CASE 1: sequential
99 if (size==1){
100     t1_ = MPI_Wtime();
101     mandelbrot_loop(Z, map, 0, xDIM*yDIM, iter);
102     t2_ = MPI_Wtime();
103 }
104 // CASE 2: parallel
105 else {
106     // distribute the data
107     int scale = sizeof(std::complex<float>) / sizeof(int);
108     if (rank==0) {
109         for (int i = 1; i < size; i++){
110             int start = xDIM*yDIM/size * i;
111             int end = xDIM*yDIM/size * (i+1); if (i==size-1) end = xDIM*yDIM;
112             MPI_Send((int *) (Z+start), (end-start)*scale, MPI_INT, i, 0,
113                     MPI_COMM_WORLD);
114         }
115         for (int i = 0; i < xDIM*yDIM/size; i++) Z_[i] = Z[i];
116     }
117     else {
118         MPI_Recv((int *) Z_, (end_idx-start_idx)*scale, MPI_INT, 0, 0,
119                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
120     }
121     // // print check
122     // printf("rank %d start_idx %d end_idx %d print %f + %fi\n",
123     //         rank, start_idx, end_idx, std::real(Z_[0]), std::imag(Z_[0]));
124
125     // start timing
126     t1_ = MPI_Wtime();
127
128     // execution
129     mandelbrot_loop(Z_, map_, 0, end_idx-start_idx, iter);
130
131     // end timing
132     t2_ = MPI_Wtime();
133
134     // gather data
135     MPI_Gather(map_, xDIM*yDIM/size, MPI_CHAR, map, xDIM*yDIM/size, MPI_CHAR, 0,
136              MPI_COMM_WORLD);
137     MPI_Barrier(MPI_COMM_WORLD);
138     // tail case
139     if (xDIM*yDIM%size != 0){
140         if (rank == size-1) MPI_Send(map_+xDIM*yDIM/size, xDIM*yDIM%size, MPI_CHAR,
141                                     0, 2, MPI_COMM_WORLD);
142         if (rank == 0) MPI_Recv(map_+xDIM*yDIM/size*size, xDIM*yDIM%size, MPI_CHAR,
143                                size-1, 2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
144     }
145     MPI_Barrier(MPI_COMM_WORLD);
146 }
147 // end timing
148 t2 = MPI_Wtime();
149
150 // end time
151 double t = t2 - t1; // overall execution time
152 double t_ = t2_ - t1_; // cpu time on calculaiton
153 double *time_arr = (double *)malloc(sizeof(double) * size);
154 double t_sum = 0;
155 MPI_Gather(&t_, 1, MPI_DOUBLE, time_arr, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
156 MPI_Barrier(MPI_COMM_WORLD);
157 for (int i = 0; i < size; i++){
158     t_sum += time_arr[i];
159 }
160 MPI_Barrier(MPI_COMM_WORLD);
161
162 // record data

```

```

158     if (rank==0 && record==1){
159         runtime_record("mpi", DIM, size, t, t_sum);
160         runtime_record_detail("mpi", DIM, size, t, time_arr);
161     }
162
163     // save png
164     if (rank==0 && save==1) mandelbrot_save("mpi", map, xDIM, yDIM);
165
166     // sync
167     MPI_Barrier(MPI_COMM_WORLD);
168
169     // end time
170     if (rank==0) runtime_print(DIM, size, t, t_sum);
171
172     // rendering
173     #ifdef GUI
174     if (rank==0){
175         // copy memory
176         map_glut = (char *)malloc(sizeof(char)*xDIM*yDIM);
177         memcpy(map_glut, map, sizeof(char)*xDIM*yDIM);
178         // plot
179         xDIM_glut = xDIM;
180         yDIM_glut = yDIM;
181         render("seq");
182         free(map_glut);
183     }
184     #endif
185
186     // free arrays
187     if (rank==0){
188         free(Z);
189         free(map);
190     }
191     free(Z_);
192     free(map_);
193     MPI_Barrier(MPI_COMM_WORLD);
194
195     // mpi finalization
196     MPI_Finalize();
197
198     return 0;
199 }

```

src/main.pthread.cpp

```

1  #include <stdio.h>
2  #include <iostream>
3  #include <fstream>
4  #include <cstdlib>
5  #include <string.h>
6  #include <chrono>
7  #include <thread>
8  #include <pthread.h>
9  #include "utils.h"
10
11
12  int main(int argc, char* argv[]) {
13      // initialization
14      float xmin = -2.0e-0;
15      float xmax = 0.6e-0;
16      float ymin = -1.3e-0;
17      float ymax = 1.3e-0;
18      int DIM = 500;
19      int save = 1;
20      int iter = 200;
21      int record = 0;
22

```

```

23 // pthread specific args
24 int nt = 1;
25
26 // parse argument
27 char buff[200];
28 for (int i = 0; i < argc; i++){
29     strcpy(buff, argv[i]);
30     if (strcmp(buff, "-n")==0 || strcmp(buff, "--ndim")==0){
31         std::string num(argv[i+1]);
32         DIM = std::stoi(num);
33     }
34     if (strcmp(buff, "-nt")==0 || strcmp(buff, "--nthread")==0){
35         std::string num(argv[i+1]);
36         nt = std::stoi(num);
37     }
38     if (strcmp(buff, "--xmin")==0){
39         std::string num(argv[i+1]);
40         xmin = std::stof(num);
41     }
42     if (strcmp(buff, "--xmax")==0){
43         std::string num(argv[i+1]);
44         xmax = std::stof(num);
45     }
46     if (strcmp(buff, "--ymin")==0){
47         std::string num(argv[i+1]);
48         ymin = std::stof(num);
49     }
50     if (strcmp(buff, "--ymax")==0){
51         std::string num(argv[i+1]);
52         ymax = std::stof(num);
53     }
54     if (strcmp(buff, "--iter")==0){
55         std::string num(argv[i+1]);
56         iter = std::stof(num);
57     }
58     if (strcmp(buff, "--save")==0){
59         std::string num(argv[i+1]);
60         save = std::stoi(num);
61     }
62     if (strcmp(buff, "--record")==0){
63         std::string num(argv[i+1]);
64         record = std::stoi(num);
65     }
66 }
67 // postprocessing
68 int xDIM = DIM;
69 int yDIM = int(DIM*(ymax-ymin)/(xmax-xmin));
70
71 // print info
72 print_info(xDIM, yDIM);
73
74 // allocation and initialization
75 std::complex<float> *Z = (std::complex<float> *)malloc(sizeof(std::complex<float>)*
    yDIM*xDIM);
76 char *map = (char *)malloc(sizeof(char) * xDIM * yDIM);
77 mandelbrot_init(Z, xDIM, yDIM, xmin, xmax, ymin, ymax);
78 Ptargs *args = (Ptargs *)malloc(sizeof(Ptargs) * nt);
79 pthread_t *threads = (pthread_t *)malloc(sizeof(pthread_t) * nt);
80
81 // start time
82 auto t1 = std::chrono::system_clock::now();
83 double *time_arr = (double *)malloc(sizeof(double)*nt);
84
85 // MAIN program
86 // create threads
87 for (int i = 0; i < nt; i++){
88     // calculate start and end index

```

```

89     int start_idx = xDIM*yDIM/nt * i;
90     int end_idx = xDIM*yDIM/nt * (i+1);
91     args[i] = (Ptargs){.Z=Z, .map=map, .start_idx=start_idx, .end_idx=end_idx, .iter
        =iter, .id=i, .time_arr=time_arr};
92     if (i==nt-1) args[i].end_idx = xDIM*yDIM;
93
94     // create independent threads
95     pthread_create(&threads[i], NULL, mandelbrot_loop_pt, (void *)(&args[i]));
96 }
97 // join threads
98 for (int i = 0; i < nt; i++){
99     pthread_join(threads[i], NULL);
100 }
101
102 // end time
103 auto t2 = std::chrono::system_clock::now();
104 auto dur = t2 - t1;
105 auto dur_ = std::chrono::duration_cast<std::chrono::duration<double>>(dur);
106 double t = dur_.count();
107 double t_sum = 0;
108 for (int i = 0; i < nt; i++) t_sum += time_arr[i];
109
110 // record data
111 if (record==1){
112     runtime_record("pth", DIM, nt, t, t_sum);
113     runtime_record_detail("pth", DIM, nt, t, time_arr);
114 }
115
116 // save png
117 if (save==1) mandelbrot_save("pth", map, xDIM, yDIM);
118
119 // end time
120 runtime_print(DIM, nt, t, t_sum);
121
122 // rendering
123 #ifdef GUI
124 // copy memory
125 map_glut = (char *)malloc(sizeof(char)*xDIM*yDIM);
126 memcpy(map_glut, map, sizeof(char)*xDIM*yDIM);
127 // plot
128 xDIM_glut = xDIM;
129 yDIM_glut = yDIM;
130 render("seq");
131 free(map_glut);
132 #endif
133
134
135 // free arrays
136 free(Z);
137 free(map);
138
139 return 0;
140 }

```

src/utils.h

```

1 #define STB_IMAGE_WRITE_IMPLEMENTATION
2 #include <stdio.h>
3 #include <iostream>
4 #include <complex>
5 #include "stb_image_write.h"
6 #include <sys/stat.h>
7 #include <sys/types.h>
8
9 #ifdef GUI
10 #include <GL/glut.h>
11 #include <GL/glu.h>

```



```

12 #include <GL/gl.h>
13
14 char *map_glut;
15 int xDIM_glut, yDIM_glut;
16 int width = 1000;
17 int xwidth, ywidth;
18 #endif
19
20 typedef struct ptargs{
21     std::complex<float> *Z;
22     char *map;
23     int start_idx;
24     int end_idx;
25     int iter;
26     int id;
27     double *time_arr;
28 } Ptargs;
29
30 void print_info(int xDIM, int yDIM){
31     printf("Name: Haoran Sun\n");
32     printf("ID: 119010271\n");
33     printf("HW: Mandelbrot Set Computation\n");
34     printf("Set xDIM to %d, yDIM to %d\n", xDIM, yDIM);
35 }
36
37 void mandelbrot_init(std::complex<float> *Z, int xDIM, int yDIM, float xmin, float xmax,
38     float ymin, float ymax){
39     for (int i = 0; i < yDIM; i++){
40         for (int j = 0; j < xDIM; j++){
41             float x = (xmax-xmin)/xDIM*j + xmin;
42             float y = (ymin-ymax)/yDIM*i + ymax;
43             // printf("%f %f\n", x, y);
44             Z[i*xDIM+j] = std::complex<float>(x, y);
45         }
46     }
47
48     char mandelbrot_iter(std::complex<float> z, std::complex<float> z0, int iter){
49         std::complex<float> p = z;
50         for (int i = 0; i < iter; i++){
51             z = z * z + z0;
52             if (std::real(z * std::conj(z)) > 4) return 255 - 255 * i/iter;
53         }
54         return 0;
55     }
56
57     void mandelbrot_loop(std::complex<float> *Z, char *map, int start_idx, int end_idx, int
58         iter){
59         for (int i = start_idx; i < end_idx; i++){
60             map[i] = mandelbrot_iter(Z[i], Z[i], iter);
61         }
62     }
63
64     void *mandelbrot_loop_pt(void *vargs){
65         // transfer args
66         Ptargs args = *(Ptargs *)vargs;
67         double *time_arr = args.time_arr;
68         int id = args.id;
69         // start time
70         auto t1 = std::chrono::system_clock::now();
71
72         // main loop
73         mandelbrot_loop(args.Z, args.map, args.start_idx, args.end_idx, args.iter);
74
75         // end time
76         auto t2 = std::chrono::system_clock::now();
77         auto dur = t2 - t1;

```

```

77     auto dur_ = std::chrono::duration_cast<std::chrono::duration<double>>(dur);
78     double t = dur_.count();
79     time_arr[id] = t;
80
81     return NULL;
82 }
83
84 void mandelbrot_save(const char *jobtype, char *map,
85     int xDIM, int yDIM){
86     char filebuff[200];
87     snprintf(filebuff, sizeof(filebuff), "mandelbrot_%s.png", jobtype);
88     stbi_write_png(filebuff, xDIM, yDIM, 1, map, 0);
89     printf("Image saved as %s.\n", filebuff);
90 }
91
92 void runtime_record(const char *jobtype, int N, int nt, double t, double t_sum){
93     const char *folder = "data";
94     mkdir(folder, 0777);
95     FILE* outfile;
96     char filebuff[200];
97     snprintf(filebuff, sizeof(filebuff), "../%s/runtime_%s.txt", folder, jobtype);
98     outfile = fopen(filebuff, "a");
99     fprintf(outfile, "%10d %5d %10.4f %10.4f\n", N, nt, t, t_sum);
100    fclose(outfile);
101    printf("Runtime added in %s.\n", filebuff);
102 }
103
104 void runtime_record_detail(const char *jobtype, int N, int nt, double t, double *
105     time_arr){
106     const char *folder = "data";
107     mkdir(folder, 0777);
108     FILE* outfile;
109     char filebuff[200];
110     snprintf(filebuff, sizeof(filebuff), "../%s/runtime_detailed_%s_%d.txt", folder,
111         jobtype, nt);
112     outfile = fopen(filebuff, "a");
113     fprintf(outfile, "%10d %5d %10.4f ", N, nt, t);
114     for (int i = 0; i < nt; i++){
115         fprintf(outfile, "%10.4f ", time_arr[i]);
116     }
117     fprintf(outfile, "\n");
118     fclose(outfile);
119     printf("Detailed runtime added in %s.\n", filebuff);
120 }
121
122 void runtime_print(int N, int nt, double t, double t_sum){
123     printf("Execution time: %.2fs, cpu time: %.2fs, #cpu %2d\n", t, t_sum, nt);
124 }
125
126 #ifndef GUI
127 void display_test(){
128     glClear(GL_COLOR_BUFFER_BIT);
129
130     glBegin(GL_POLYGON);
131     glVertex2f(0, 0);
132     glVertex2f(1, 0);
133     glVertex2f(1, 1);
134     glVertex2f(0, 1);
135     glEnd();
136
137     glFlush();
138 }
139
140 void plot(){
141     // display test
142     // initialization

```

```

142     glClear(GL_COLOR_BUFFER_BIT);
143     glColor3f(0.0f, 0.0f, 0.0f);
144
145     // draw points
146     GLfloat pointSize = 1.0f;
147     glPointSize(pointSize);
148     glBegin(GL_POINTS);
149     glClear(GL_COLOR_BUFFER_BIT);
150     for (int i = 0; i < yDIM_glut; i++){
151         for (int j = 0; j < xDIM_glut; j++){
152             int c0 = (unsigned char) map_glut[i*xDIM_glut+j];
153             float c = c0;
154             c = c0 / 255.0;
155             glColor3f(c, c, c);
156             glVertex2f(j, yDIM_glut-i);
157         }
158     }
159     glEnd();
160
161     // flush
162     glFlush();
163 }
164
165 void resize(int x, int y){
166     glutReshapeWindow(xwidth, ywidth);
167 }
168
169 void render(const char *jobtype){
170     // glu init
171     int glufoo = 1;
172     char q[] = " ";
173     char *glubar[1];
174     glubar[0] = q;
175     glutInit(&glufoo, glubar);
176     glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
177
178     // set x and y width
179     xwidth = width;
180     ywidth = yDIM_glut*width/xDIM_glut;
181     glutInitWindowSize(xwidth, ywidth);
182     glutCreateWindow(jobtype);
183     glMatrixMode(GL_PROJECTION);
184     gluOrtho2D(0, xDIM_glut, 0, yDIM_glut);
185
186     // display func
187     glutDisplayFunc(plot);
188     // glutDisplayFunc(display_test);
189     glutReshapeFunc(resize);
190
191     glutMainLoop();
192 }
193
194 #endif
195

```