Plot Mandelbrot Set using Openmp, Pthread and MPI

Xin Huang
Dept. of CST, THU
ID: 2011011253

September 8, 2013

Abstract

The Mandelbrot set is a very famous pattern of fraction. Mathematically, the mandelbrot set is a set of points whose boundary is distinctive.

I wrote the mandelbrot set implementation program in three different parallel approach – *OpenMP* , *pthread* and *MPI* . The program supports kinds of settings and present the plot in gtk.

and present the plot in both gray and colored way, with interactive user interface which leads to convenient exploration of the fratal.

The design of the program, then efficiency analysis based on running result on clusters, and the image coloring algorithm will be furtherly discussed in article.

This is *homework 3* for course *Parallel Programming*

Keyword Mandelbrot set, fractal, parallel

Contents

1 Instruction to Run the Program

You can simply follow $make- > make\ run$ to run the program. For more detailed instruction, seed Appedix **??**

2 Intro to Mandelbrot set

We define a sequence for each point c on the complex plane:

$$z_0 = 0 \tag{1}$$

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

Mandelbrot set is defined the set of complex numbers which satisfy:

$$\exists M \in \mathbb{R}, s.t \lim_{n\to\infty} |z_n| < M$$

Calculation is based on the following theorem:

For a complex number c, if $c \in M$, then $|c| \leq 2$

We call $n = \min(\{m : |z_m| > 2\})$ the **Escape Time** of a specific complex number c. In practice ,we set up a upper bound of iteration, name **n_iter_max**. A complex number of **Escape Time** exceed **n_iter_max** will be considered in M

3 Design & Approach

3.1 Overview

The program contains three parts: the mandelbrot function part, graph render part, gtk display part. **The Mandelbrot Function** is the definition of the transition function. In this program, the function calculation is done in the mandelbrot function. Give the result of each iteration.

Graph Renderer(GR) deals with the work of render. This part includes render configuration and render result. The render config determin the width and height the program renders and the domain as well as the number of worker. The render result store the image that rendered by the worker. **Worker** contains three methods to finish the task, representing number of instances used for computing. For *pthread* and *OpenMP* , it is the number of threads, for *MPI* , it is the number of nodes. The coloring algorithm based on [**color_algo**] by *Francisco Garcia, Angel Fernandez, Javier Barrallo* and *Luis Martin* is also implemented. **User Interface** is used **GTK+-2.0**

3.2 *OpenMP*

It is simple to use openmp to parallel the serial program. After setting the config and setting the task, the program invoke the loop to calculate and render the result. During this time, we can use openmp to paralleling.

3.3 *pthread*

It is a little more complex to implement in pthread. The program makes the taskpool to manage the rendering tasks. The threads won't stop fetching the task from the task pool until the task pool is empty. The taskpool sets the configuration and uses the mutex to lock the fetching function to prevent the race condition when get the tasks and modify the total number of the tasks. After getting the task from the task pool, every thread use $\text{pthread}_create function to call the render function. Then call the pthread_join to wait$

3.4 *MPI*

It is a little painful to write a mpi program in this problem. The processes need to send the configuration and the result to each other, and it needs to know the position to render. The process 0 is the master and firstly it sends the configuration to others. Other processes send the results to the master processor. The master processor is responsible for the distribution.

Routine for master process:

1. if all render tasks are collected, goto step 7
2. broadcast render configuration among all processes
3. receive render finishing signal from slave processes, along with tasks previously assigned
4. to see if render result from a slave should be collected. If so, communicate with corresponding slave process, and fetch result.
5. ask **Task Scheduler** for new task assignment.
   (a) If a new task assignment obtained, send new assignment to that slave process
   (b) otherwise send render phase termination signal to savle process
6. back to 1
7. quit render procedure.

Routine for slave process:

1. fecth new render configuration from master process.
2. if it is a abort singal, goto step 9
3. set current task to none
4. report render result of current task to master process
5. fetch new task from master process
6. if no new tasks available, goto step 1
7. deal with current render task
8. goto step 4
9. quit render procedure.

In a short summary, master process deal with render scheduling and UI, slave processes are just render machines under the hood.

3.5 Coloring Algorithm

I use google to find a satisfying coloring algorithm.
We introduce a correction term in index used for coloring. Previously we defined the index

$$n = \textbf{EscapeTime}$$

, now we subtract a extra term $\log\log|z_n| * C$ (or other similar term that contains information of z_n), where C is a contant, lead n becomes a real number

$$n = \textbf{EscapeTime} - \log\log|z_n| * C$$

then we change the fix-sized palette to a continuous real function which map this number to RGB colorspace(range in $[0, 1]$). Here I used

$$\begin{cases} red = \dfrac{n}{n\_iter\_max} \\ green = \dfrac{\cos(0.003 * n) + 1}{2} \\ blue = \dfrac{\sin(0.003 * n) + 1}{2} \end{cases}$$

to make the image greenish and bluish, produce a 'cool' feel. Using this formula will produde a smooth color gradation rather than a step to step color scheme.

3.6 User Interface

One can use mouse button click to control the plotting position and scale by clicking

- **Left button**

  Zoom in with the point clicked located in the center

- **Right button**

  Zoom out with the point clicked located in the center

- **Middle button**

  make point clicked located in the center

The zoom rate can be specified by command line option, see Appendix **??** for detailed explanation.

4 Result & Analysis

All programs use the same function to calculate the **Escape Time**. The curve is of a certain size of the image (both width and height, in pixel).

All programs are to render a region on complex plane with left-botom coordinate $(-1.5, -1)$ and size of $2x2$

4.1 *OpenMP* as Backend

	256	512	1024	2048	4096	8192	16384
2	122	199	353	658	1270	2493	4937
4	60	102	178	333	637	1250	2473
6	46	74	122	224	430	837	1653
8	53	77	112	178	325	638	1242
10	42	71	83	147	279	526	1003
12	49	53	73	124	226	429	835
1	213	368	676	1286	2513	4965	9864

Table 1: Data-Processors Time(ms) table for openmp

Efficiency of *OpenMP* is calculated by fomula below:

$$E = \frac{Execution\ time\ using\ one\ thread}{Execution\ time\ using\ multi-threads \times number\ of\ threads}$$

As *OpenMP* is a simple and general tool to construct threaded application, its overhead on thread management may not that optimal. It may not correctly infer which variable is shared and must be locked when use. For my program, non of the variables except for the loop variable is shared. In case here, *OpenMP* just happen to mess up.

Additionally, at first I used the static method to distribute the load and the performance was worse than the pthread. Then I change the method to dynamic, which can make the tasks more balance for each processors and then the performance was perfectly better than the static method. It indicated that the work is not balance in each prossessor. In fact the amount of the calculation in the middle is much more than in the edge. So in paralleling programming, it's also crucial to consider the balance in each processor.

The **Speedup** is good in the dynamic method. When number of iter equals to 16384, the speedup reach 11.81 when 12 processors. The efficieny is 0.98.

4.2 *pthread* as Backend

	256	512	1024	2048	4096	8192	16384
2	104	181	334	641	1253	2478	4934
4	53	93	167	326	646	1283	2560
6	36	64	115	220	435	866	1726
8	29	54	100	183	359	728	1396
10	25	41	74	142	276	553	1092
12	27	39	86	193	277	638	1143

Table 2: Data-Processors Time(ms) table for pthread

Efficiency of *pthread* is calculated by fomula same as *OpenMP*. With dynamical task scheduling mechanism, the efficiency of threaded program almost have no loss in efficiency when number of processors raise up. The low cost of creating a new thread may also be counted in.

But the limit to threaded program is the number of processors in a single computer. Although it has a high efficieny, it can not be extended to run on clusters directectly.

4.3 *MPI* as Backend

	2048	4096	8192	16384	32768	65536
12	239	475	941	1868	3735	7465
24	136	265	516	1013	2007	4029
36	102	174	341	706	1348	2680
48	102	180	352	670	1370	2674
60	108	173	340	672	1371	2669

Table 3: Data-Processors Time(ms) table for mpi

Because of master-slave structure of the program, efficiency of *MPI* is calculated by fomula below:

$$E = \frac{Execution\ time\ using\ two\ processors}{Execution\ time\ using\ a\ multiprocessor \times (number\ of\ processors - 1)}$$

The lack of efficiency when size of image is small is due to the high overload on inter-process communication and *MPI* library itself, which count a huge part in running time. The efficiency rise up only when the size of image is big enough so that the calculation of determining the **Mandelbrot set** is enormous.

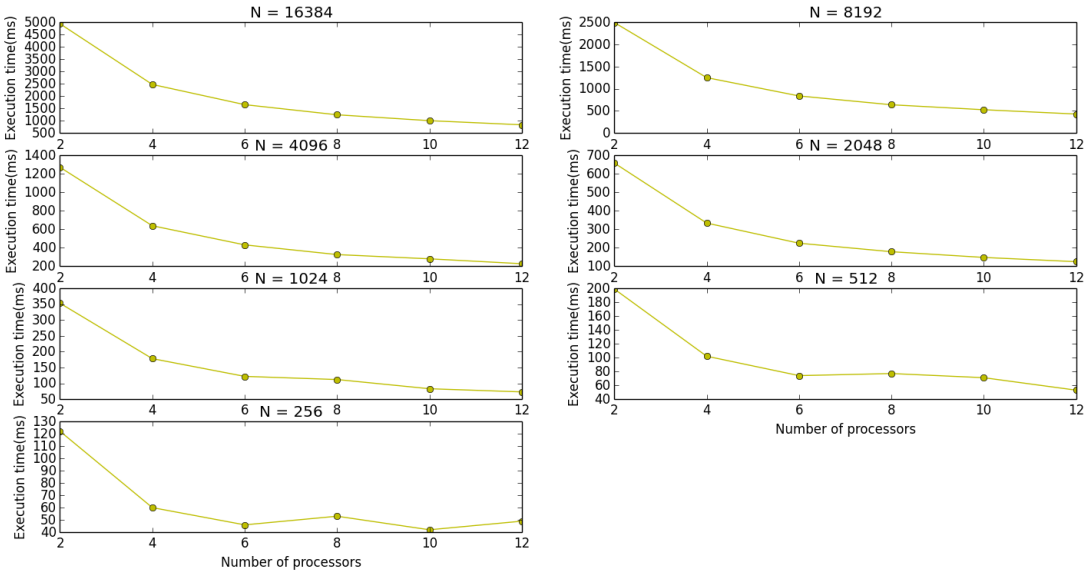
4.4 Strong Scalability

- **Strong Scalability** means the problem size is fixed while the number of processes are increased. In strong scalability, if the speedup is equal to number of processes the program will be considered as linear scale. However, it's not very possible to achieve this goal according to the Amdahl's law. In this program, when the number of processes *nworker* increases, the numbers in each process will be less. And the percentage of communication will be larger. So the speedup will decrease. And the results illustrated support the idea. The more processes there are, the more cost of the communication, which decreases the strong scalability and the cost of time of waiting will be larger.

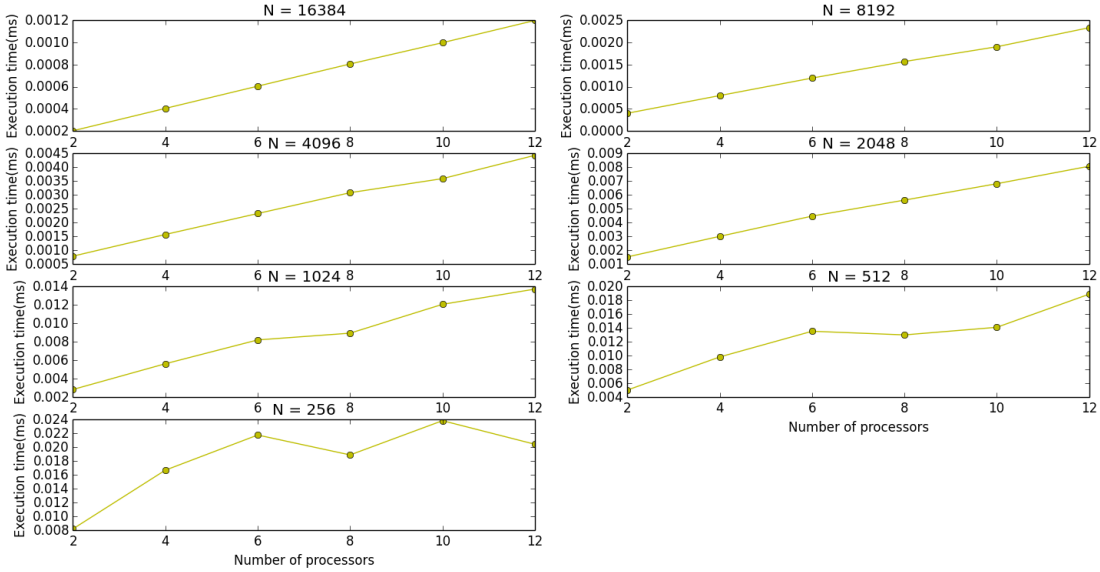
  In the large data set($niter$ = 65536, 32768, 16384 , 8192) the -1/ x graphs are linear as illustration. While in the small data set($niter$ = 4096, 2048, 1024 , 512) the -1/x graphs are not so good(when $nworker > 36$) , which implicates that the strong scalability .

5 Figure

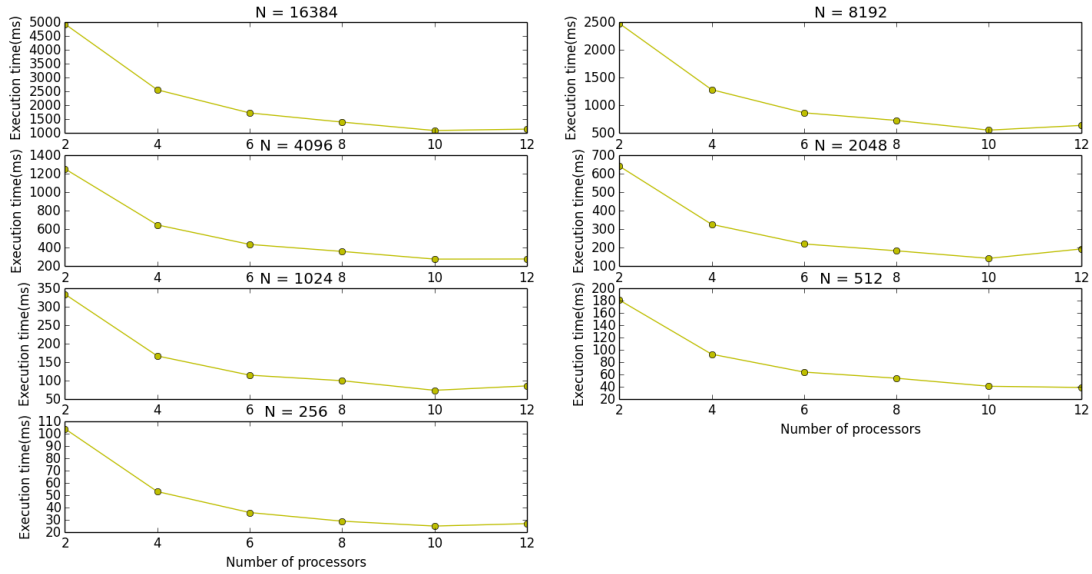
openmp graph:



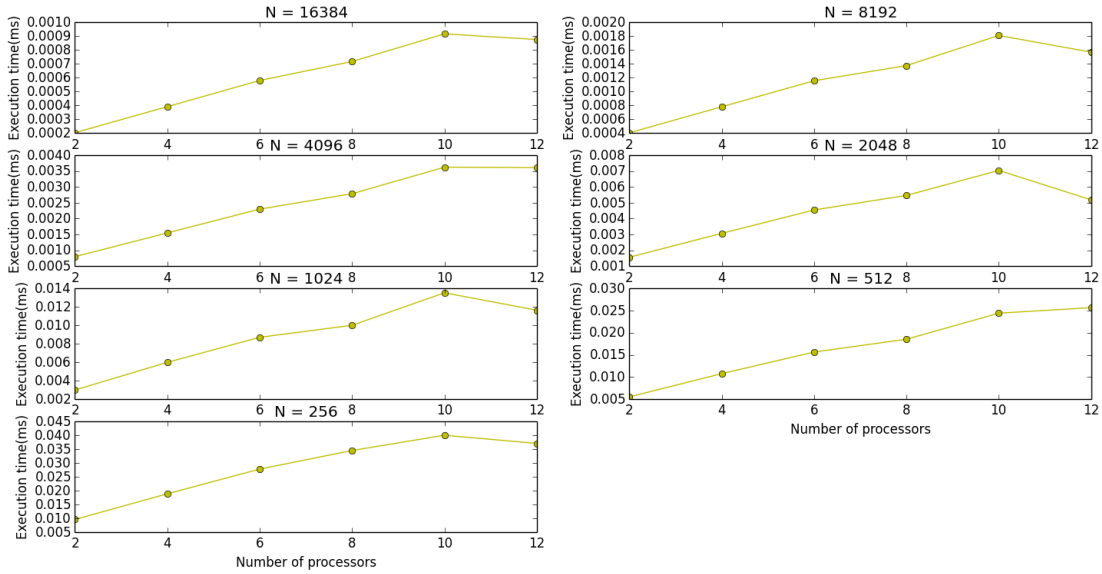
openmp speedup graph:



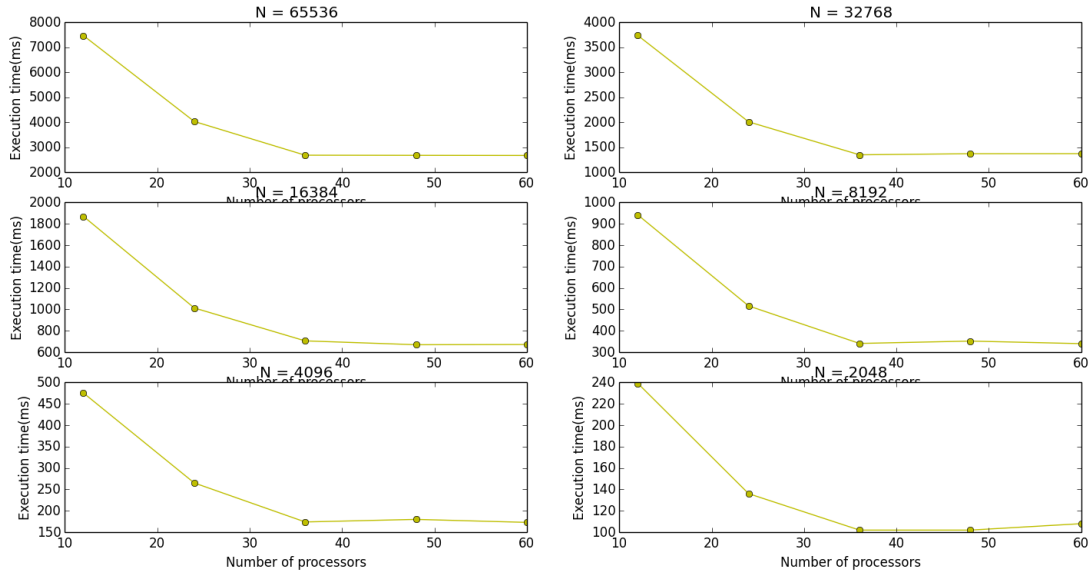
pthread graph:



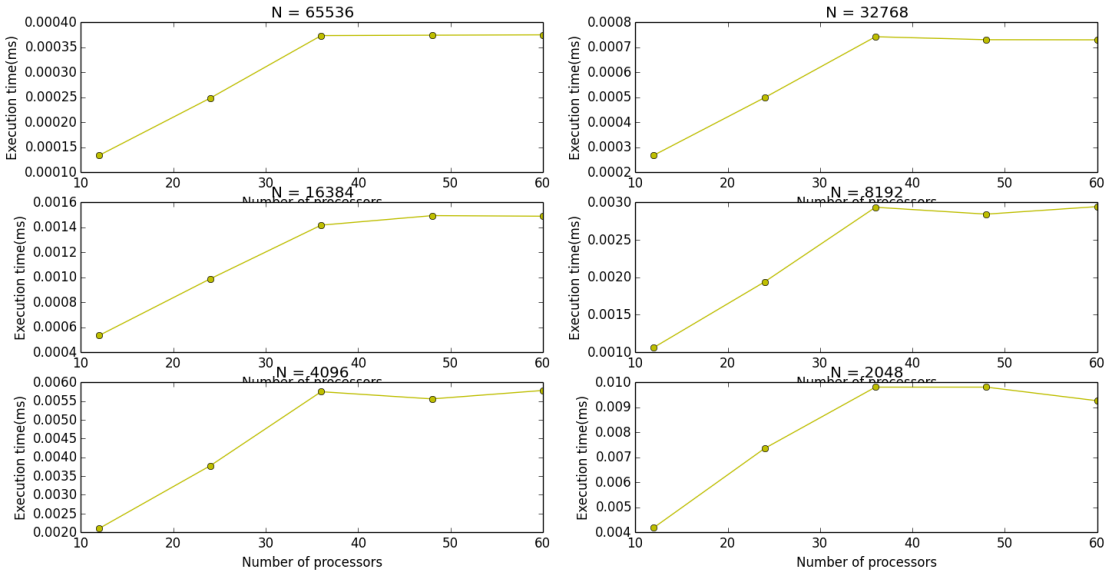
pthread speedup graph:



mpi graph:

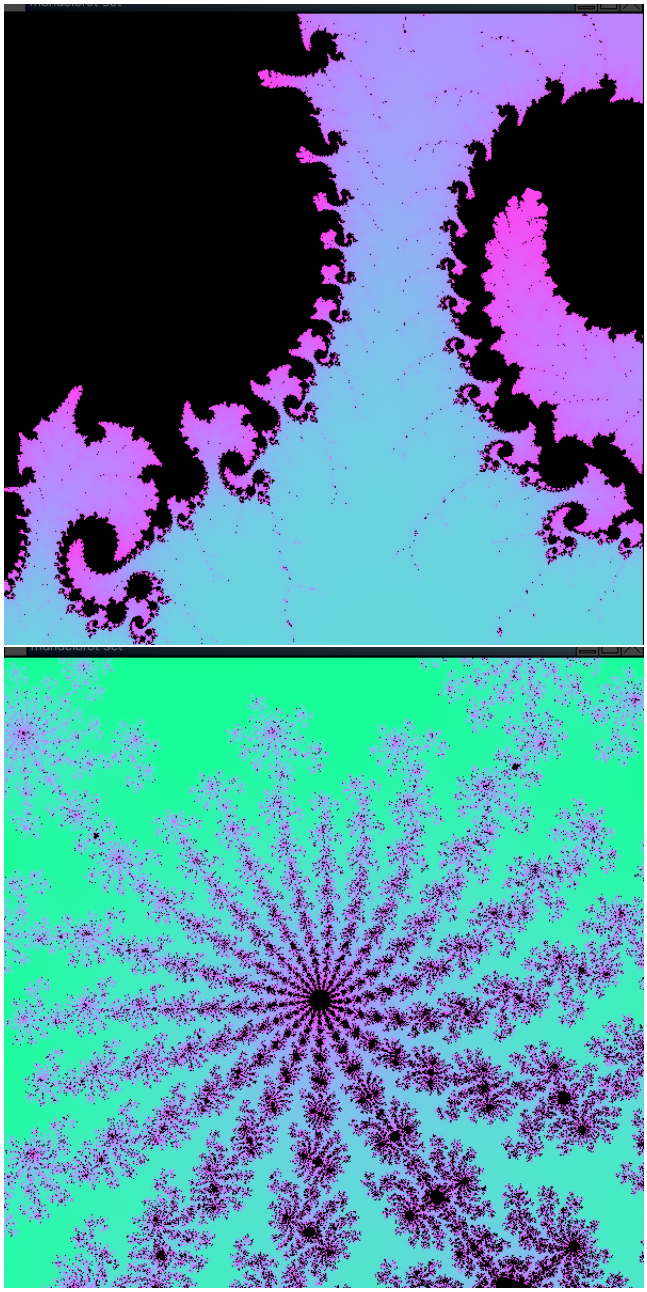


mpi speedup graph:



6 Experience

This problem is much more complex but amazing than the previous works. The mandelbrot set illustrates the beauty of Mathematics. It is very tricky to write a program to plot the mandelbrot shape.



7 Screenshots

8 Source Code

8.1 Worker

../../src/openmp.hh

```
1  #ifndef __OPENMP__
2  #define __OPENMP__
3
4  #include"render.hh"
5  #include<cstdio>
6
```

```
class OpenmpRender: public GraphRender{
    public:
        virtual GraphRenderRes *render(Func *func, const GraphRenderConf &conf);
};

#endif
```

../../src/openmp.cc

```
#include"openmp.hh"
#include"timer.hh"


GraphRenderRes *OpenmpRender::render(Func *func, const GraphRenderConf &conf){
    const Rect &domain = conf.domain;
    int npixel = conf.width * conf.height;
    RenderTask *task = new RenderTask[npixel];
    GraphRenderRes *res = NULL;
    if (conf.is_res == true)
        res = new GraphRenderRes(conf.width, conf.height);
    int p = 0;
    for (int i = 0; i < conf.width; i++){
        for (int j = 0; j < conf.height; j++, p++){
            double x = domain.x + (double) i/conf.width * domain.width;
            double y = domain.y + (double) j/conf.height *domain.height;
            task[p].num = Complex(x, y);
            int t = i * conf.height + (conf.height-j-1);
            if (!conf.is_res){
                task[p].cpixel = NULL;
                task[p].gpixel = NULL;
            }
            else{
                task[p].cpixel = res->rgb->data+t;
                task[p].gpixel = res->rgb->data+t;
            }
        }

    }

    Timer timer;
    timer.set_start();

#pragma omp parallel for num_threads(conf.nworker) schedule(dynamic, 1)
    for (int i = 0; i < npixel; i++){
        func->render(task[i]);
    }

    int time = timer.get_time();
    printf("time: %d\n", time);

    return res;

}
```

../../src/pthread.hh

```
#ifndef __PTHREAD__
#define __PTHREAD__

#include<pthread.h>
#include<cstdio>
#include<string.h>
#include"render.hh"

#define NTF_MAX_DEFAULT 10000

class TaskPool{
```

```
12      public:
13          pthread_mutex_t m_mutex;
14
15          RenderTask *task;
16          int taskcnt, ntask;
17          TaskPool(const GraphRenderConf &conf, GraphRenderRes *res){
18              memset(&m_mutex, 0, sizeof(m_mutex));
19              int npixel = conf.width*conf.height;
20              task = new RenderTask[npixel];
21              ntask = npixel;
22              taskcnt=0;
23              const Rect &domain = conf.domain;
24              for (int i = 0, p = 0; i < conf.width; i++)
25                  for (int j = 0; j < conf.height; j++, p++){
26
27                      task[p].num.real = domain.x + (double)i/conf.width*domain.width;
28                      task[p].num.imag = domain.y + (double)j/conf.height*domain.height;
29                      if (!conf.is_res){
30                          task[p].cpixel = NULL;
31                          task[p].gpixel = NULL;
32                      }
33                      else{
34                          int dpos = i * conf.height + (conf.height-j-1);
35                          task[p].cpixel = res->rgb->data+dpos;
36                          task[p].gpixel = res->gray->data+dpos;
37                      }
38                  }
39          }
40          virtual double progress(){
41              return (double) taskcnt / ntask;
42          }
43
44          virtual int fetch_task(int nnum, RenderTask *&mem){
45              pthread_mutex_lock(&m_mutex);
46              int n = ntask - taskcnt; //task remained
47              if (nnum <= n) n = nnum; //if enough
48              mem = task + taskcnt; //next pos
49              taskcnt+=n;
50              pthread_mutex_unlock(&m_mutex);
51              return n;
52          }
53          virtual ~TaskPool(){delete [] task;}
54  };
55
56  class PthreadRender : public GraphRender{
57      public:
58          int m_ntf_max;
59
60          virtual void dorender(Func *func, TaskPool *taskpool);
61          friend void *render_call(void *arg);
62
63  //        PthreadRender(int ntf_max = NTF_MAX_DEFAULT):m_ntf_max(ntf_max){}
64          PthreadRender(){m_ntf_max = NTF_MAX_DEFAULT;}
65          virtual GraphRenderRes *render(Func *func, const GraphRenderConf &conf);
66  };
67
68  struct PthreadArg{
69      PthreadRender *render;
70      TaskPool *taskpool;
71      Func *func;
72  };
73
74  #endif
```

../../src/pthread.cc

```
1   #include<cstring>
2   #include"pthread.hh"
```

```cpp
#include"timer.hh"
#include<cstdio>

void *render_call(void *arg){
    PthreadArg *parg = static_cast<PthreadArg*>(arg);
    parg->render->dorender(parg->func, parg->taskpool);
    pthread_exit(NULL);
}

GraphRenderRes *PthreadRender::render(Func *func, const GraphRenderConf &conf){
    int nthread = conf.nworker;

    GraphRenderRes * res = NULL;
    if (conf.is_res == true){
        res = new GraphRenderRes(conf.width, conf.height);
    }

    TaskPool *taskpool = new TaskPool(conf, res);
    pthread_t *threads = new pthread_t[nthread];
    PthreadArg *parg = new PthreadArg[nthread];

    Timer timer;
    timer.set_start();

    for (int i = 0; i < nthread; i++){
        PthreadArg *arg = parg + i;
        arg->render = this;
        arg->taskpool = taskpool;
        arg->func = func;
        pthread_create(&threads[i], NULL, render_call, arg);
    }

    for (int i = 0; i < nthread; i++)
        pthread_join(threads[i], NULL);
    int time = timer.get_time();
    printf("time: %d\n", time);

    delete taskpool;
    delete[] threads;
    delete[] parg;
    return res;
}

void PthreadRender::dorender(Func *func, TaskPool *taskpool){
    RenderTask *tasks;
    int ntask, tottask = 0;
    while((ntask = taskpool->fetch_task(m_ntf_max, tasks))){
        tottask += ntask;
        for (int i = 0; i < ntask; i++)
            func->render(tasks[i]);
    }
}
```

../../src/mpi.hh

```cpp
#ifndef __MPIR__
#define __MPIR__


#include<mpi.h>
#include"render.hh"

struct TaskSche{
    int tasknum, maxtnum, taskcnt;
    TaskSche(int tasknum, int maxtnum):tasknum(tasknum), maxtnum(maxtnum), taskcnt(0){}
    void fetch_task(int &start, int &end){
        int n = tasknum-taskcnt;
        if (n > maxtnum) n = maxtnum;
```

```
14          start = taskcnt;
15          end = taskcnt+n;
16          taskcnt = end;
17      };
18      bool finished(){return tasknum == taskcnt;}
19  };
20
21  class MPIRender : public GraphRender{
22      public:
23
24          int npro, pro_id;
25
26          MPI_Datatype MPI_TYPE_COLOR;
27          MPI_Datatype MPI_TYPE_GRAPHRENDERCONF;
28
29          MPI_Status status;
30
31          int maxtnum;
32
33          int npixel;
34
35          GraphRenderConf conf;
36
37          TaskSche *ts;
38          MPI_Request send_request;
39          int pixels_finished;
40          GraphRenderRes *res;
41
42          void task_dis();
43
44          struct Task{
45              int start, end;
46              ColorRGB *rgb, *gray;
47              Task():rgb(NULL){}
48          };
49
50          void send_result(const Task &task);
51          void task_get(Task &task);
52
53          void send_new_conf(const GraphRenderConf &conf);
54          bool recv_new_conf(GraphRenderConf &conf);
55
56      public:
57          MPIRender(int argc, char *argv[], int maxtnum);
58          ~MPIRender();
59          virtual GraphRenderRes *render(Func *func, const GraphRenderConf &conf);
60          int rank() const {return pro_id;}
61  };
62
63  #endif
```

../../src/mpi.cc

```
1   #include<cstdlib>
2   #include<mpi.h>
3   #include<cmath>
4   #include<algorithm>
5   #include"mpi.hh"
6   #include<cassert>
7   #include"timer.hh"
8
9
10  #define TAG_APPLY 1
11  #define TAG_NEW 2
12  #define TAG_JOB_FINISHED 3
13  #define TAG_IMG 4
14
15
```

```
16
17  MPIRender::MPIRender(int argc, char* argv[], int maxtnum){
18      this->maxtnum = maxtnum;
19      MPI_Init(&argc, &argv);
20      MPI_Comm_size(MPI_COMM_WORLD, &npro);
21      if (npro == 1){
22          exit(0);
23      }
24      MPI_Comm_rank(MPI_COMM_WORLD, &pro_id);
25
26      MPI_Type_contiguous(sizeof(ColorRGB), MPI_CHAR, &MPI_TYPE_COLOR);
27      MPI_Type_contiguous(sizeof(GraphRenderConf), MPI_CHAR, &MPI_TYPE_GRAPHRENDERCONF);
28
29      MPI_Type_commit(&MPI_TYPE_COLOR);
30      MPI_Type_commit(&MPI_TYPE_GRAPHRENDERCONF);
31  }
32
33  MPIRender::~MPIRender(){
34      if (pro_id == 0){
35          GraphRenderConf conf;
36          conf.nworker = -1;
37          send_new_conf(conf);
38      }
39      MPI_Finalize();
40  }
41
42  void MPIRender::task_dis(){
43      int range[2];
44      MPI_Recv(range, 2, MPI_INT, MPI_ANY_SOURCE, TAG_JOB_FINISHED, MPI_COMM_WORLD, &status);
45
46      int worker = status.MPI_SOURCE;
47      if (range[0] != range[1]){
48          assert(range[1] > range[0]);
49          if (conf.is_res){
50              ColorRGB *data;
51
52              data = res->rgb->data + range[0];
53
54              MPI_Recv(data, range[1] - range[0], MPI_TYPE_COLOR, worker, TAG_IMG, MPI_COMM_WORLD, &status);
55
56          }
57          pixels_finished += range[1] - range[0];
58      }
59      ts->fetch_task(range[0], range[1]);
60
61      MPI_Send(range, 2, MPI_INT, worker, TAG_NEW, MPI_COMM_WORLD);
62  }
63
64  void MPIRender::send_result(const MPIRender::Task &task){
65      int range[2] = {task.start, task.end};
66      MPI_Send(range, 2, MPI_INT, 0, TAG_JOB_FINISHED, MPI_COMM_WORLD);
67      if (conf.is_res && task.start != task.end){
68          MPI_Send(task.rgb, task.end - task.start, MPI_TYPE_COLOR, 0, TAG_IMG, MPI_COMM_WORLD);
69      }
70  }
71
72  void MPIRender::task_get(MPIRender::Task &task){
73      int range[2];
74      MPI_Recv(range, 2, MPI_INT, 0, TAG_NEW, MPI_COMM_WORLD, &status);
75      task.start = range[0];
76      task.end = range[1];
77  }
78
79  GraphRenderRes *MPIRender::render(Func *func, const GraphRenderConf &iconf){
80      res = NULL;
81      GraphRenderConf tconf = iconf;
82      Timer timer;
83
```

```
84      this->conf = tconf;
85      if (pro_id == 0){
86          send_new_conf(conf);
87          if (conf.is_res)
88              res = new GraphRenderRes(conf.width, conf.height);
89          npixel = conf.width*conf.height;
90          maxtnum = std::max(10000, (int)floor(ceil((double)npixel / npro)));
91          maxtnum = std::min(maxtnum, 1000000);
92          ts = new TaskSche(npixel, maxtnum);
93          pixels_finished = 0;
94
95          timer.set_start();
96          while(pixels_finished != npixel)
97              task_dis();
98          timer.get_time();
99
100         if (conf.is_res){
101             for (int i = 0; i < conf.width; i++){
102                 for (int j = 0; j < conf.height/2; j++){
103                     int p0 = i * conf.height + j;
104                     int p1 = i * conf.height + (conf.height - 1 - j);
105                     std::swap(res->rgb->data[p0], res->rgb->data[p1]);
106                     std::swap(res->rgb->data[p0], res->rgb->data[p1]);
107                 }
108             }
109         }
110         delete ts;
111
112     }
113
114     else {
115         res = NULL;
116         int alloc_pixel = 0;
117         Task task;
118         while (recv_new_conf(conf)){
119             npixel = conf.width * conf.height;
120             task.start = task.end = -1;
121             if (conf.is_res && npixel > alloc_pixel){
122                 alloc_pixel = npixel;
123                 if (task.rgb) delete task.rgb;
124                 if (task.gray) delete task.gray;
125                 task.rgb = new ColorRGB[npixel];
126                 task.gray = new ColorRGB[npixel];
127
128             }
129             const Rect &domain = conf.domain;
130             RenderTask rtask;
131
132             while(1){
133                 send_result(task);
134                 task_get(task);
135
136                 if (task.start == task.end)
137                     break;
138
139                 int i = task.start / conf.height;
140                 int j = task.start % conf.height;
141                 int p = task.start;
142                 if (!conf.is_res){
143                     rtask.gpixel = NULL;
144                     rtask.cpixel = NULL;
145                 }
146                 for (int dpos = 0; p < task.end; p++, dpos++){
147                     rtask.num.real = domain.x + (double) i/conf.width*domain.width;
148                     rtask.num.imag = domain.y + (double) j/conf.height*domain.height;
149                     if (conf.is_res){
150                         rtask.gpixel = task.gray + dpos;
151                         rtask.cpixel = task.rgb + dpos;
```

```
                    }
                    func->render(rtask);
                    j++;
                    if (j == conf.height){
                        i++;
                        j = 0;
                    }
                }
            }
        }
        if (conf.is_res)
        {
            delete task.gray;
            delete task.rgb;
        }
    }
    return res;
}


void MPIRender::send_new_conf(const GraphRenderConf &conf){
    GraphRenderConf buf = conf;
    MPI_Bcast(&buf, 1, MPI_TYPE_GRAPHRENDERCONF, 0, MPI_COMM_WORLD);
}

bool MPIRender::recv_new_conf(GraphRenderConf &conf){
    MPI_Bcast(&conf, 1, MPI_TYPE_GRAPHRENDERCONF, 0, MPI_COMM_WORLD);
    if (conf.nworker == -1)
        return false;
    return true;
}
```

../../src/render.hh

```
#ifndef __RENDER__
#define __RENDER__

#include<cstdlib>
#include<cstdio>
#include"rect.hh"
#include"func.hh"
#include"image.hh"

#define REAL_T_FMT "lf"

struct GraphRenderConf{
    int width; int height;
    Rect domain;
    int nworker;
    bool is_res;

    GraphRenderConf(){}
    GraphRenderConf(int width, int height, const Rect &domain, int nworker, bool is_res = true) : width(width), height
    void show(){
        fprintf(stderr, "image size: %dx%d\n", width, height);
        fprintf(stderr, "domain: (%.18" REAL_T_FMT ",%.18" REAL_T_FMT ")\n        [%.18" REAL_T_FMT ",%.18" REAL_T_FMT
        fprintf(stderr, "nworker: %d\n", nworker);
    }


};



struct GraphRenderRes{
    Image *rgb, *gray;
    GraphRenderRes():rgb(NULL), gray(NULL){}
    GraphRenderRes(int width, int height){
```

```
35          rgb = new Image(width, height);
36          gray = new Image(width, height);
37      }
38      ~GraphRenderRes(){
39          if (rgb)
40              delete rgb;
41          if (gray)
42              delete gray;
43      }
44
45  };
46
47  class GraphRender{
48      public:
49          virtual GraphRenderRes *render(Func *func, const GraphRenderConf &conf)=0;
50          virtual ~GraphRender(){}
51  };
52  #endif
```

8.2 Gtk main

../../src/gtk.hh

```
1   #include"paint.hh"
2   #include"comp.hh"
3   #include"image.hh"
4   #include"render.hh"
5   #include"rect.hh"
6   #include"image.hh"
7
8   #include<cstdio>
9   #include<cstdlib>
10  #include<string>
11  #include<string.h>
12  #include<cairo.h>
13  #include<gtk/gtk.h>
14  #include<gdk-pixbuf/gdk-pixbuf.h>
15
16
17  GdkPixbuf *img2pixbuf(Image *img);
18
19  #define ST_GRAY 0
20  #define ST_RGB 1
21
22  using namespace std;
23
24  struct RenderConfig{
25      GraphRender *rdr;
26      Func *func;
27      GraphRenderConf gc;
28      GraphRenderRes *output;
29      GdkPixbuf *gpixbuf, *cpixbuf;
30      GdkPixbuf *pixbuf;
31
32      int width(){return output->rgb->width;}
33      int height(){return output->rgb->width;}
34
35      int show_type;
36      void set_show_type(int type){
37          pixbuf = cpixbuf;
38          show_type = type;
39      }
40
41      RenderConfig(){
42          memset(this, 0, sizeof(RenderConfig));
43          show_type = ST_RGB;
```

```
    }

    void unref(){
        if (gpixbuf)
            g_object_unref(gpixbuf);
        if (cpixbuf)
            g_object_unref(cpixbuf);
    }

    ~RenderConfig(){
        delete rdr;
        delete func;
        if (output) delete output;
        unref();
    }

    bool render(bool to_pix=false){
        if (output) delete output;
        if (cpixbuf){
            g_object_unref(cpixbuf);
            cpixbuf = NULL;
            pixbuf = cpixbuf;
        }
        if (gpixbuf){
            g_object_unref(gpixbuf);
            gpixbuf = NULL;
            pixbuf = gpixbuf;
        }

        gc.show();

        output = rdr->render(func, gc);

        if (to_pix)
            this->to_pixbuf();

        if (output == NULL)
            return false;
        return true;
    }

    void to_pixbuf(){
        if (output){
            unref();
            cpixbuf = img2pixbuf(output->rgb);
            gpixbuf = img2pixbuf(output->gray);
            pixbuf = cpixbuf;
        }
    }
};


RenderConfig rcfg;

static gboolean delete_event(
        GtkWidget *,
        GdkEvent *,
        gpointer
        ){
    return FALSE;
}


static void destroy(GtkWidget *, gpointer){
    gtk_main_quit();
}


```

```
112 GdkPixbuf *img2pixbuf(Image *img){
113     GdkPixbuf *pixbuf = gdk_pixbuf_new(GDK_COLORSPACE_RGB, false, 8, img->width, img->height);
114     int n_channels = gdk_pixbuf_get_n_channels(pixbuf);
115     int rowstride = gdk_pixbuf_get_rowstride(pixbuf);
116 
117     guchar *pixel = gdk_pixbuf_get_pixels(pixbuf);
118 
119     for (int i = 0; i < img->width; i++){
120         for (int j = 0; j < img->height; j++){
121             guchar *p = pixel + j *rowstride + i * n_channels;
122             ColorRGB *c = img->data + i * img->height + j;
123             p[0] = c->r * 255;
124             p[1] = c->g * 255;
125             p[2] = c->b * 255;
126         }
127     }
128     return pixbuf;
129 }
130 
131 
132 static gboolean da_expose_callback(
133         GtkWidget *widget,
134         GdkEventExpose *,
135         gpointer
136         ){
137     GdkPixbuf *pix = rcfg.pixbuf;
138     cairo_t *cr = gdk_cairo_create(widget->window);
139     gdk_cairo_set_source_pixbuf(cr, pix, 0, 0);
140     cairo_paint(cr);
141     cairo_destroy(cr);
142     return FALSE;
143 }
144 
145 static gboolean cb_timeout(GtkWidget *widget){
146     if (widget->window == NULL)
147         return FALSE;
148 
149     gtk_widget_queue_draw_area(widget, 0, 0, widget->allocation.width, widget->allocation.height);
150     return TRUE;
151 }
152 
153 double zoom_scale = 2.0;
154 
155 static gboolean cb_clicked(GtkWidget *, GdkEventButton *event, gpointer){
156     static const unsigned LEFT_BUTTON = 1,
157                   MIDDLE_BUTTON = 2,
158                   RIGHT_BUTTON = 3;
159     if (event->button == LEFT_BUTTON || event->button == RIGHT_BUTTON || event->button == MIDDLE_BUTTON){
160         int cx = event -> x;
161         int cy = event -> y;
162 
163         GraphRenderConf &gc = rcfg.gc;
164         Rect &dm = gc.domain;
165 
166         double rx = dm.x + (double)cx / gc.width * dm.width;
167         double ry = dm.y + (double)(gc.height - cy) / gc.height * dm.height;
168         if (event->button == LEFT_BUTTON)
169             dm.width /= zoom_scale, dm.height /= zoom_scale;
170         else if (event->button == RIGHT_BUTTON)
171             dm.width *= zoom_scale, dm.height *= zoom_scale;
172 
173         dm.x = rx - dm.width / 2;
174         dm.y = ry - dm.height / 2;
175 
176         rcfg.render(true);
177     }
178     return TRUE;
179 }
```

```
180
181
182
183 void image_init(int argc, char* argv[]){
184     gtk_init(&argc, &argv);
185 }
186
187 void image_show(){
188     GtkWidget *window;
189
190     int border_width = 0;
191     int window_width = rcfg.width()+border_width*2;
192     int window_height = rcfg.height()+border_width*2;
193     window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
194     gtk_container_set_border_width(GTK_CONTAINER(window), border_width);
195     gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);
196     gtk_window_set_default_size(GTK_WINDOW(window), window_width, window_height);
197     gtk_widget_set_size_request(window, window_width, window_height);
198
199     GtkWidget *da = gtk_drawing_area_new();
200     gtk_widget_set_size_request(da, rcfg.width(), rcfg.height());
201
202     g_signal_connect(window, "delete-event", G_CALLBACK(delete_event), NULL);
203     g_signal_connect(window, "destroy", G_CALLBACK(destroy), NULL);
204     gtk_widget_add_events(da, GDK_BUTTON_PRESS_MASK);
205     g_signal_connect(da, "button-press-event", G_CALLBACK(cb_clicked), NULL);
206     g_signal_connect(da, "expose_event", G_CALLBACK(da_expose_callback), NULL);
207     g_timeout_add(1000/30, (GSourceFunc)cb_timeout, da);
208     gtk_container_add(GTK_CONTAINER(window), da);
209     gtk_widget_show_all(window);
210
211     gtk_main();
212 }
213
214 void img2ppm(Image *image, string filename){
215     FILE *fout;
216     if (filename == "-")
217         fout = stdout;
218     else fout = fopen(filename.c_str(), "w");
219
220     fprintf(fout, "P6\n%d %d\n255\n", image->width, image->height);
221     unsigned char pixel[3];
222     for (int i = 0; i < image->height; i++){
223         for (int j = 0; j < image->width; j++){
224             ColorRGB *c = image->data + j*image->height + i;
225             pixel[0] = c->r*255;
226             pixel[1] = c->g*255;
227             pixel[2] = c->b*255;
228             fwrite(pixel, sizeof(pixel), 1, fout);
229         }
230     }
231
232     if (fout != stdout)
233         fclose(fout);
234 }
235
236 void img2pgm(Image *image, string filename){
237     FILE *fout;
238     if (filename == "-")
239         fout = stdout;
240     else fout = fopen(filename.c_str(), "w");
241
242     fprintf(fout, "P5\n%d %d\n225\n", image->width, image->height);
243     unsigned char pixel;
244     for (int i = 0; i < image->height; i++){
245         for (int j = 0; j < image->width; j++){
246             ColorRGB *c = image->data + j * image->height + i;
247             pixel = c->r*255;
```

```
248             fwrite(&pixel, sizeof(pixel), 1, fout);
249         }
250     }
251
252     if (fout != stdout)
253         fclose(fout);
254 }
255
256 template<typename T>
257 void check_lower_bound(T &num, T lower_bound){
258     if (num < lower_bound)
259         num = lower_bound;
260 }
261
262 int str2num(std::string str){
263     int ret = 0, sign = 1, start = 0;
264     if (str[0] == '-')
265     {    sign = -1; start = 1;}
266     for (int i = start; i < str.length(); i++)
267         ret = ret*10+str[i]-'0';
268     return ret*sign;
269
270 }
271
272 double str2real(string str){
273     double ret;
274     sscanf(str.c_str(), "%lf", &ret);
275     return ret;
276 }
```

../../src/main.cc

```
1  #include<cstdio>
2  #include<cstdlib>
3  #include<string>
4  #include<getopt.h>
5  #include<unistd.h>
6
7  #include"mandelbrot.hh"
8  #include"gtk.hh"
9  #include"rect.hh"
10 #include"render.hh"
11 #include"mpi.hh"
12 #include"openmp.hh"
13 #include"pthread.hh"
14
15
16 GraphRender *GraphRenderFactory(int category, int argc, char*argv[]){
17     switch(category){
18         case 0:
19             return new MPIRender(argc, argv, 100000);
20             break;
21         case 1:
22             return new OpenmpRender();
23             break;
24         case 2:
25             return new PthreadRender();
26             break;
27     }
28     exit(0);
29 }
30
31 char* name;
32
33
34
35 int main(int argc, char* argv[]){
36     name = argv[0];
```

```
37
38  //    int width = atoi(argv[1]), height = atoi(argv[2]);
39  //    int niter_max = atoi(argv[3]);
40      int width = 600, height = 600, niter_max = 2048;
41
42      bool show = true;
43
44      int parallel_method = 1;
45
46      int nworker = sysconf(_SC_NPROCESSORS_ONLN);
47
48      option options[] = {
49          {"nworker", required_argument, NULL, 'n'},
50          {"niter", required_argument, NULL, 'i'},
51          {"type", required_argument, NULL, 't'}
52      };
53      int opt;
54      while ((opt = getopt_long(argc, argv, "t:n:i:", options, NULL)) != -1){
55          switch(opt){
56              case 't':
57                  parallel_method = atoi(optarg);
58                  break;
59              case 'i':
60                  niter_max = atoi(optarg);
61                  break;
62              case 'n':
63                  nworker = atoi(optarg);
64          }
65
66      }
67
68      bool is_res = true;
69
70      rcfg.gc.domain.x = 1000;
71      rcfg.gc.domain.y = 1;
72
73      rcfg.rdr = GraphRenderFactory(parallel_method, argc, argv);
74      rcfg.func = new Mandelbrot(Setting(TERM_FLAG_NITER, niter_max));
75      Rect rect(-1, -1, 2, 2);
76      rcfg.gc = GraphRenderConf(width, height, rect, nworker, is_res);
77      if(!rcfg.render())
78          return 0;
79
80      image_init(argc, argv);
81      rcfg.to_pixbuf();
82
83  //    if (is_res == true && rcfg.output){
84  //        img2ppm(rcfg.output->rgb, coutput);
85  //    }
86      if (show)
87          image_show();
88
89  }
```

8.3 Function

../../src/func.hh

```
1   #ifndef __FUNC__
2   #define __FUNC__
3
4   #include"comp.hh"
5   #include"image.hh"
6
7   #define PM_GRAY 0
8   #define PM_COLORFUL 1
```

```

struct RenderTask{
    Complex num;
    int niter;
    ColorRGB *gpixel;
    ColorRGB *cpixel;
    RenderTask(){
        num = Complex(0, 0);
    }
};

class Func{
    public:
        bool iscolor;
        virtual void paint_mode(bool color){
            iscolor = color;
        }

        virtual void render(RenderTask &task)=0;
        virtual ~Func(){}
};

#endif
```

../../src/mandelbrot.hh

```
#define TERM_FLAG_NITER 1
#define TERM_EPS_DEFAULT 1e-15
#define TERM_ITER_DEFAULT 200

#include"paint.hh"
#include"func.hh"

struct Setting{
    int term_flag;
    int max_iter_n;
    double eps;
    Setting(int term_flag, int max_iter_n = TERM_ITER_DEFAULT, double eps = TERM_EPS_DEFAULT):term_flag(term_flag), ma
};

class Mandelbrot:public Func{

    public:

        Setting m_setting;
        Paint *paint_colorful;

        Mandelbrot(const Setting &setting = Setting(TERM_FLAG_NITER));
        virtual ~Mandelbrot();
        virtual void render(RenderTask &task);
};
```

../../src/mandelbrot.cc

```
#include"mandelbrot.hh"

#include<cmath>
#include<cstdlib>

Mandelbrot::Mandelbrot(const Setting &setting):m_setting(setting){
    int color = 8192;
    paint_colorful = new Paint(color);
}

Mandelbrot::~Mandelbrot(){
    delete paint_colorful;
}
```

```
14
15  void Mandelbrot::render(RenderTask &task)
16  {
17      static const ColorRGB black(0, 0, 0),
18                    white(1, 1, 1);
19      Complex &c = task.num;
20
21      double x = c.real, y = c.imag;
22
23      int niter = m_setting.max_iter_n;
24      for (double x_sqr, y_sqr;
25              ((x_sqr = x*x) + (y_sqr = y*y)) <= 4 && niter;niter--){
26          y = 2*x*y+c.imag;
27          x = x_sqr-y_sqr+c.real;
28      }
29      if (task.cpixel == NULL || task.gpixel == NULL)
30          return;
31      if (x*x+y*y > 4){
32
33          double dist = (m_setting.max_iter_n - niter) - log(log(x*x+y*y))/M_LN2;
34          *task.gpixel = ColorRGB((sin(0.08*dist)+1)*0.5);
35          *task.cpixel = ColorRGB(
36                  dist/m_setting.max_iter_n,
37                  (cos(0.03*dist)+1)*0.5,
38                  (sin(0.03*dist)+1)*0.5
39                  );
40
41      }
42      else{
43          *task.gpixel = black;
44          *task.cpixel = black;
45      }
46  }
```

8.4 Others

../../src/timer.hh

```
1   #include<sys/time.h>
2
3
4   class Timer{
5       public:
6           timeval t1, t2;
7           void set_start(){
8               gettimeofday(&t1, NULL);
9           }
10          void set_stop(){
11              gettimeofday(&t2, NULL);
12          }
13          int get_time(){
14              gettimeofday(&t2, NULL);
15              int res = (t2.tv_sec - t1.tv_sec) * 1000 + (t2.tv_usec - t1.tv_usec) / 1000;
16              return res;
17          }
18  };
```

../../src/rect.hh

```
1   #ifndef __RECT__
2   #define __RECT__
3
4   #include"comp.hh"
5
6   class Rect{
7       public:
```

```
8           double x, y, width, height;
9
10          Rect(){}
11
12          Rect(double x, double y, double width, double height):x(x), y(y), width(width), height(height){}
13  };
14
15  #endif
```
