Plot Mandelbrot Set using Openmp, Pthread and MPI

Xin Huang
Dept. of CST, THU
ID: 2011011253

September 5, 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 A

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 $\text{pthread}_c reate function to call the render function. Then call the pthread_j oin to wait$

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.

Figure 1: The right-bottom figures in both 2x2 and 2x3 grid are rendered using advance coloring algorithm as described above, remaining figures are simple palette driven coloring algorithm with palette size from 512 to 8192

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 A 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	132	241	458	891	1754
3488	6947				
4	87	162	315	617	1222
2432	4852				
6	64	117	225	440	872
1741	3463				
8	50	102	187	347	689
1369	2726				
10	44	96	149	289	587
1145	2252				
12	44	66	125	242	482
940	1877				

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.

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

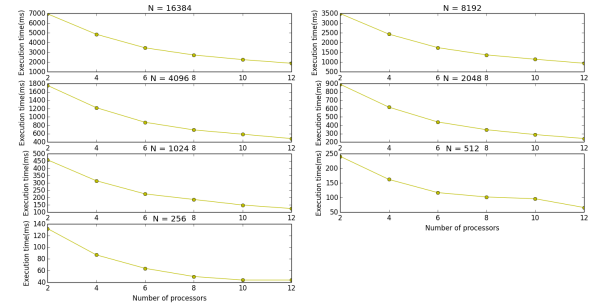
Table 3: Data-Processors Time(ms) table for mpi

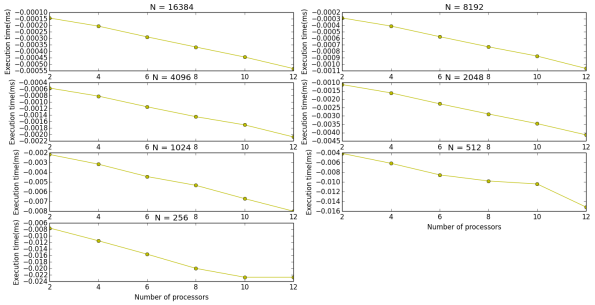
Because of master-slave structure of the program, efficiency of *MPI* is calculated by fomula below:

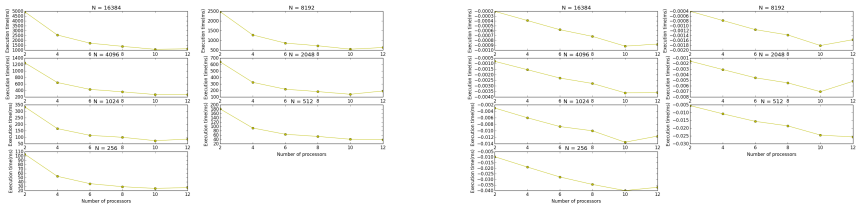
$$E = \frac{Execution\ time\ using\ two\ processors}{Execution\ time\ using\ a\ multiprocessor \times (number\ of\ processors - 1)}$$

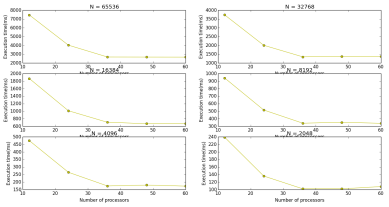
The lack of efficiency when size of image is small is due to the high overload on inter-process communication and *MPI* library itself, which count a huge part in running time. The efficiency rise up only when the size of image is big enough so that the calculation of determining the **Mandelbrot set**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

5 Figure









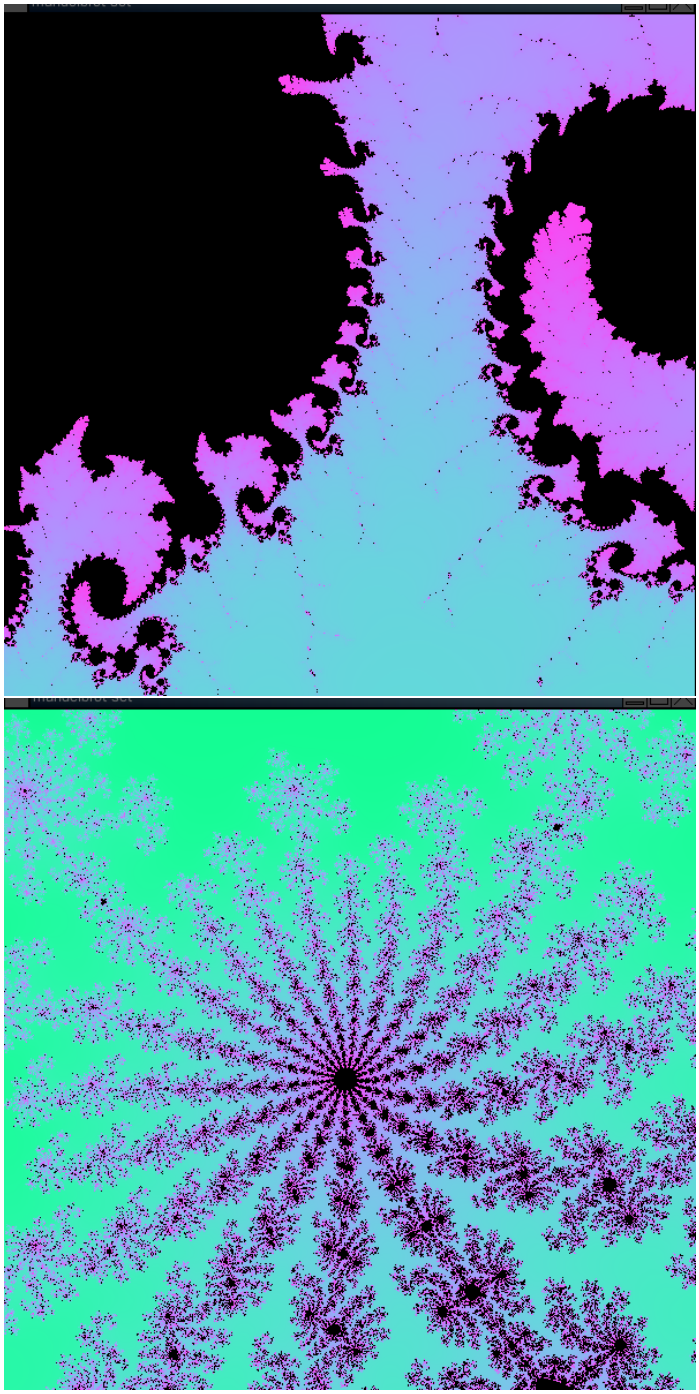
6 Screenshots

7 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.

8 Source Code

../src/openmp.hh ../src/openmp.cc ../src/pthread.hh ../src/pthread.cc ../src/mpi.hh ../src/mpi.cc



A Command Line Help

res/help