The Chinese University of Hong Kong, Shenzhen

CSC4005

Distributed and Parallel Computing

Report for CSC4005 Project 2

Author:

Feng Yutong

Student Number:

120090266

October 26, 2022

Contents

1 Introduction

This project implements two versions of Mandelbrot set computation. One is MPI version and the other is multi-thread version.

Mandelbrot set is the set of points in a complex plane that are quasi-stable when computed by iterating function, which is given by:

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

where z_{k+1} is the (k+1)th iteration of the complex number $z = a + bi$ and c is a complex number given the position of the point in the complex plane.

This report will investigate into the following aspects:

1. Performance comparison between sequential, MPI and pthread version.

2. Performance comparison between parallel computing under different problem size.

3. Performance comparison between different core/thread numbers under the same problem size.

2 Method

2.1 Pthread version

Step 1. Use *pthread_create* to create a thread pool, bind each thread with function *worker* and arguments such as thread rank and thread number.

Step 2. Each thread compute equal amount of data, which is given by total_size / thread_number. Since thread share global variable *Point *data*. They can get their part of data by calculating with the thread rank. Notice the calculation of each point is mutually exclusive, so there is no need to use *pthread_mutex*.

Step 3. Join all thread and draw graph if with GUI.

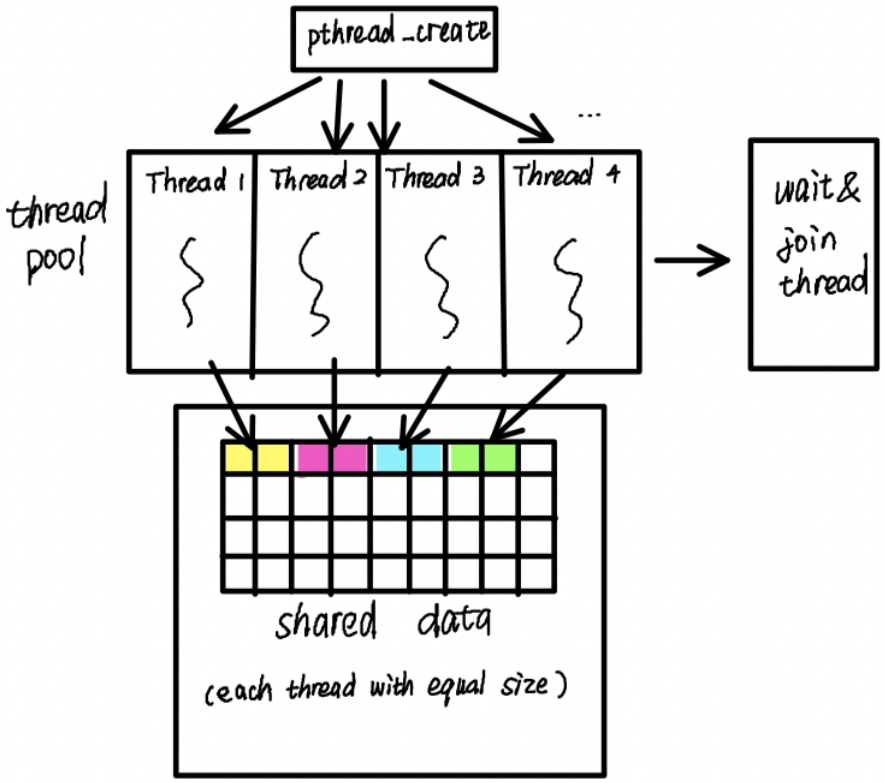


Figure 1: Pthread flow

2.2 MPI version

Step 1. MPI initialization. Use *MPI_Type_struct* and *MPI_COMMIT* to construct the struture *Point* in MPI datatype. Then call *MPI_Scatter* function to distribute the same amount of data to the sub-processes. The distributed amount can be calculated as total_size / world_size.

Step 2. Each sub-process computes its data.

Step 3. Call *MPI_Gather* to collect the point information to the root process.

Step 4. Root process draws graph if with GUI.

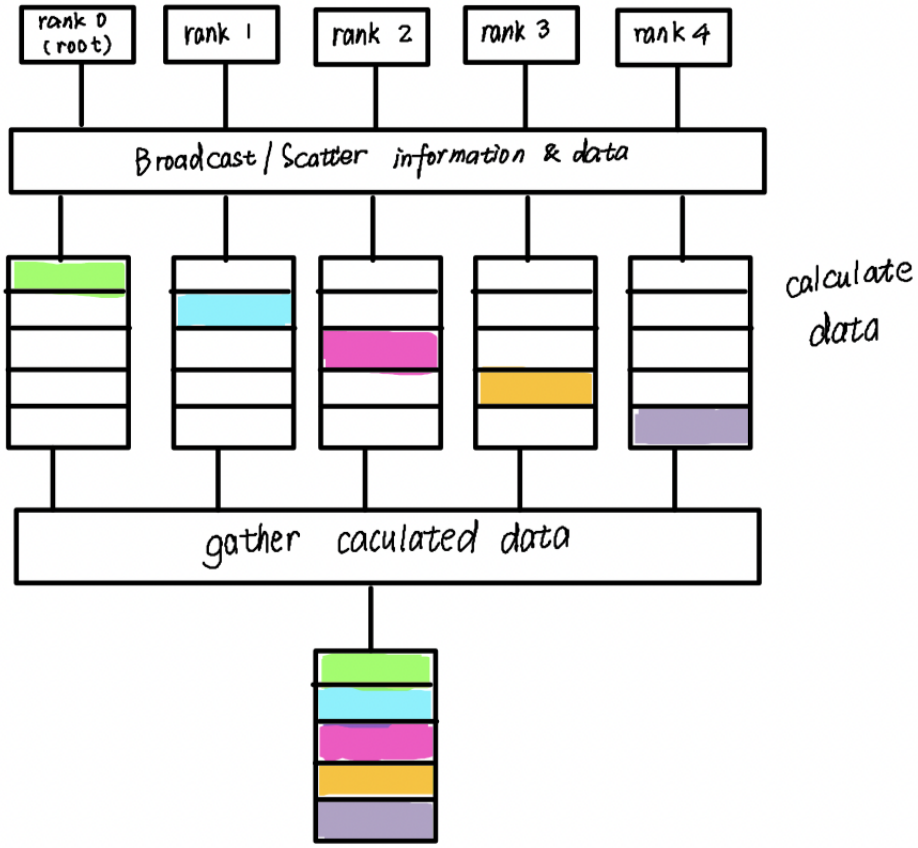


Figure 2: MPI flow

2.3 Algorithm analysis

The problem size is determined by X_RESN, Y_RESN, max_iteration. Suppose $N = \text{X\_RESN} \times Y\_RESN$ and max_iteration is M. We assume the time to compute one point is $O(M)$

1. Sequential version:

The program computes points sequentially. Thus, time complexity is $O(MN^2)$.

2. MPI version:

Suppose the number of processes is p. Each sub-process computes $\frac{N^2}{p}$ data. The program also takes $O(N^2)$ to scatter data and $O(N^2)$ to gather data. Thus, total time complexity is $O(\frac{MN^2}{p} + N^2)$

3. Pthread version:

Suppose the number of thread is p. Each thread computes $\frac{N^2}{p}$ data. The program also takes $O(p)(p << N)$ to create threads. Since threads requires no mutex, there's no need to wait others. And data is stored in global buffer so there's no need to pass data. Thus, total time complexity is $O(\frac{MN^2}{p})$.

3 Result

3.1 Execution

Compile:

```
// MPI without GUI
mpic++ mpi.cpp -o mpi -std=c++11

// MPI with GUI
mpic++ -I/usr/include -L/usr/local/lib -L/usr/lib -
    lglut -lGLU -lGL -lm mpi.cpp -o mpig -DGUI -std=c
    ++11

// pthread without GUI
g++ pthread.cpp -lpthread -o pthread -O2 -std=c++11
```

```
9
10 // pthread with GUI
11 g++ -I/usr/include -L/usr/local/lib -L/usr/lib -lglut -
       lGLU -lGL -lm -lpthread pthread.cpp -o pthreadg -
       DGUI -O2 -std=c++11
```

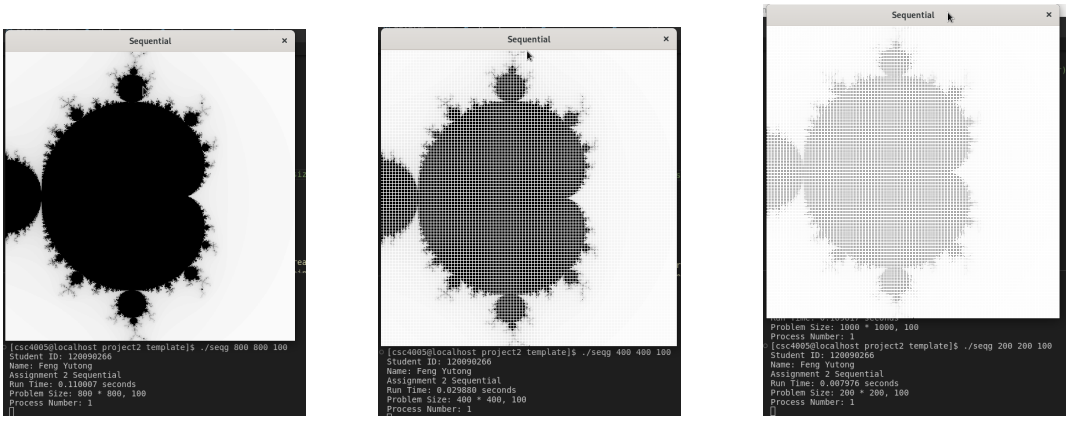
Run:

```
1 X_RESN means the resolution of x axis, Y_RESN means the
       resolution of y axis.
2 max_iteration is a parameter of Mandelbrot Set
       computation.
3 n_proc is the number of processed of MPI.
4 n_thd is the number of threads of pthread.
5
6 // Sequential
7 ./seq $X_RESN $Y_RESN $max_iteration
8 ./seqg $X_RESN $Y_RESN $max_iteration
9
10 // MPI
11 mpirun -np $n_proc ./mpi $X_RESN $Y_RESN $max_iteration
12 mpirun -np $n_proc ./mpig $X_RESN $Y_RESN
       $max_iteration
13
14 // pthread
15 ./pthread $X_RESN $Y_RESN $max_iteration $n_thd
16 ./pthreadg $X_RESN $Y_RESN $max_iteration $n_thd
```

Or you can **modify** the *pthread_run.sh, mpi_run.sh, seq˙run.sh* and use *qsub* to run multiple processes together.

3.2 Demo

Here shows GUI output of three different image size (small: 200 * 200, middle: 400 * 400, big: 800 * 800) with a max_iteration of 100 by running sequential, MPI and Pthread respectively. The same output indicates the correctness of the algorithm implementation. Since drawing takes much more time, the data sizes for drawing are chosen relatively small.



Sequential (800*800) Sequential (400*400) Sequential (200*200)

Figure 3: Sequential output of different size

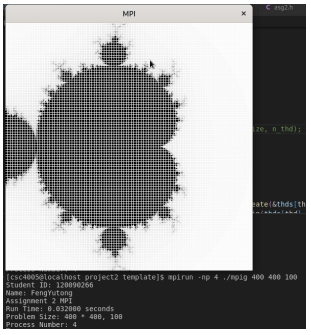
3.3 Performance analysis

3.3.1 Test Design

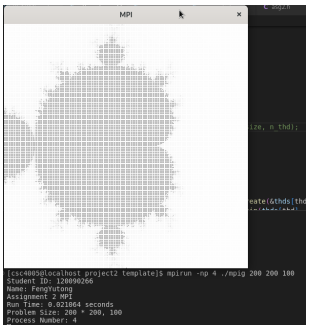
max_iteration: 100

Data size (N = X_RESN × Y_RESN): samll(360k, 1000k, 4000k), big(32000k, 64000k, 10000k)

MPI (800*800)

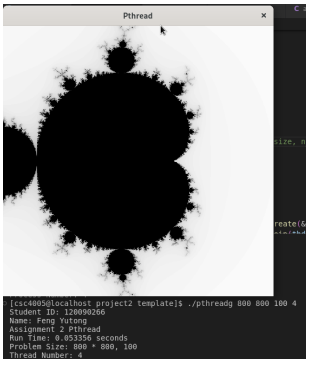


MPI (400*400)

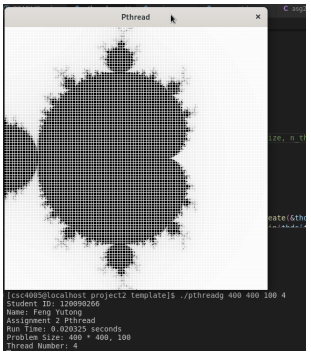


MPI (200*200)

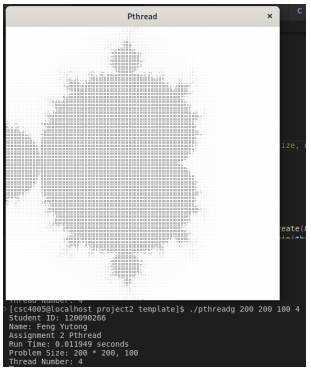
Figure 4: MPI output of different size



Pthread (800*800)



Pthread (400*400)



Pthread (200*200)

Figure 5: Pthread output of different size

Core/Thread Number: 1, 2, 4, 8, 16, 32

Data is the average of three runs.

3.3.2 Sequential vs MPI vs Pthread

In this program, when core (thread) number is one, MPI (Pthread) works in the same way as sequential version. Figure 6-7 and Table 1 show that three version share similar rate under different problem size.

	Sequential	MPI	Pthread
360k	0.10922	0.099276	0.108878
1000k	0.274349	0.272874	0.275174
4000k	1.05206	1.089958	1.066666
32000k	8.382444	8.962498	8.407762
64000k	16.796947	17.946627	16.817923
10000k	26.270319	28.079346	26.272907

Table 1: Sequential vs MPI vs Pthread

However, Figure 8 shows different result. With larger core (thread) number, pthread performs better than MPI. This may due to the communication overhead. MPI needs scatter and gather data while pthread calculates within a shared buffer. This corresponds to our previous analysis that it takes $O(N^2)$ for MPI to do communication.

3.3.3 Time

From Figure 9-12 and Table 2-3, we can see that the runtime decreases with the increasing of the core (thread) number as we analysed before. However, decreasing rate will decay when the process number is large. For MPI, when process number is too large, communication overhead has a more significant impact on the total

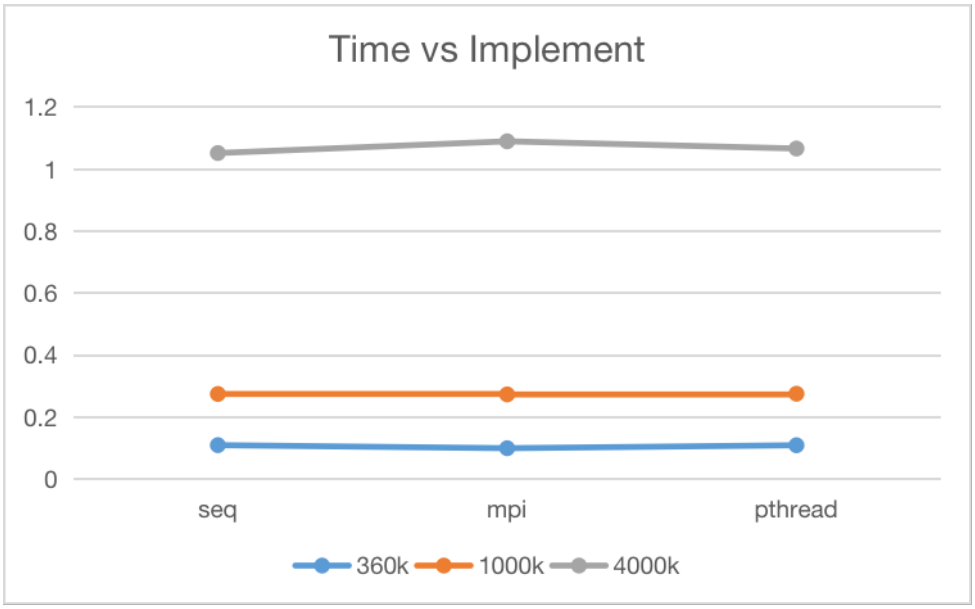


Figure 6: Sequential vs MPI vs Pthread (small size)

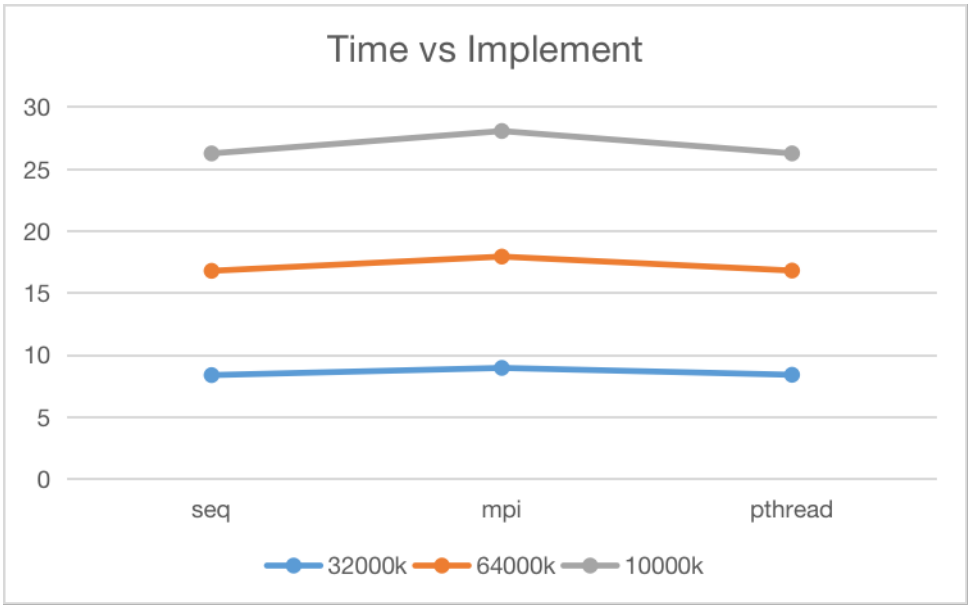


Figure 7: Sequential vs MPI vs Pthread (big size)

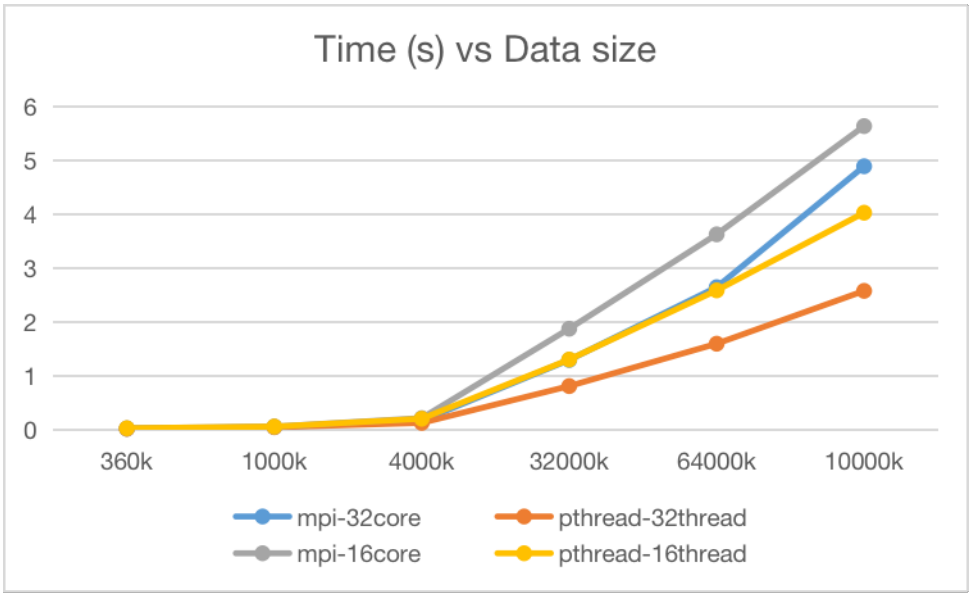


Figure 8: MPI vs Pthread

runtime. For pthread, when the number of threads is more than that of the physical threads, the processor may need to divide the work into small time slice and switch between threads. This mechanism guarantees concurrent execution but brings overhead to switch and restore. We can amplify the impact by calculating speedup and efficiency.

MPI	1	2	4	8	16	32
360k	0.099276	0.079655	0.059009	0.0358	0.030992	0.023898
1000k	0.272874	0.198608	0.140533	0.090979	0.063195	0.055172
4000k	1.089958	0.780102	0.538682	0.339037	0.220112	0.172284
32000k	8.962498	6.396177	4.325927	2.728381	1.882113	1.301732
64000k	17.946627	12.918007	8.621031	5.461302	3.63581	2.65368
10000k	28.079346	20.076817	13.99323	8.362227	5.647872	4.901152

Table 2: Runtime of MPI

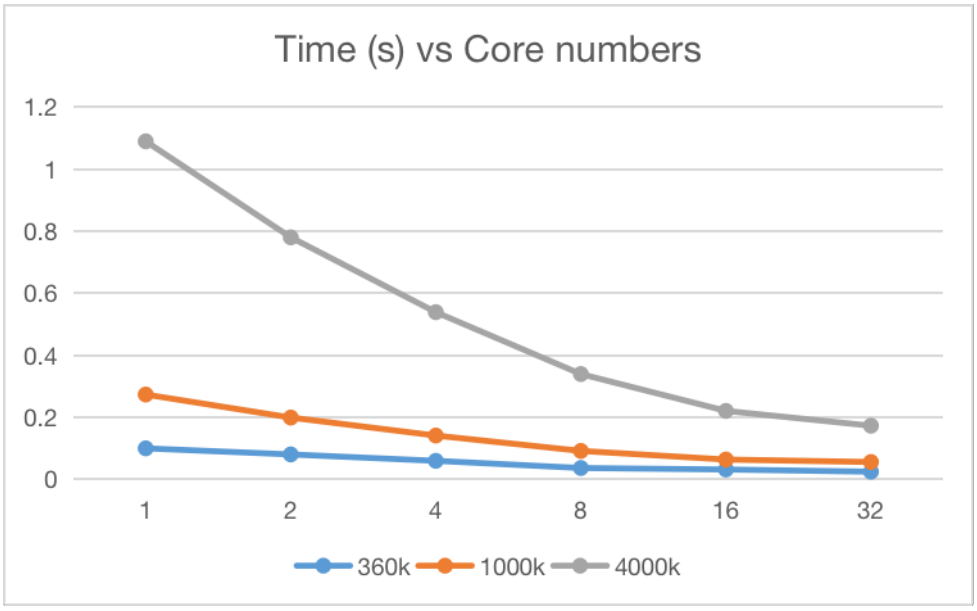


Figure 9: Time vs Core number (small size)

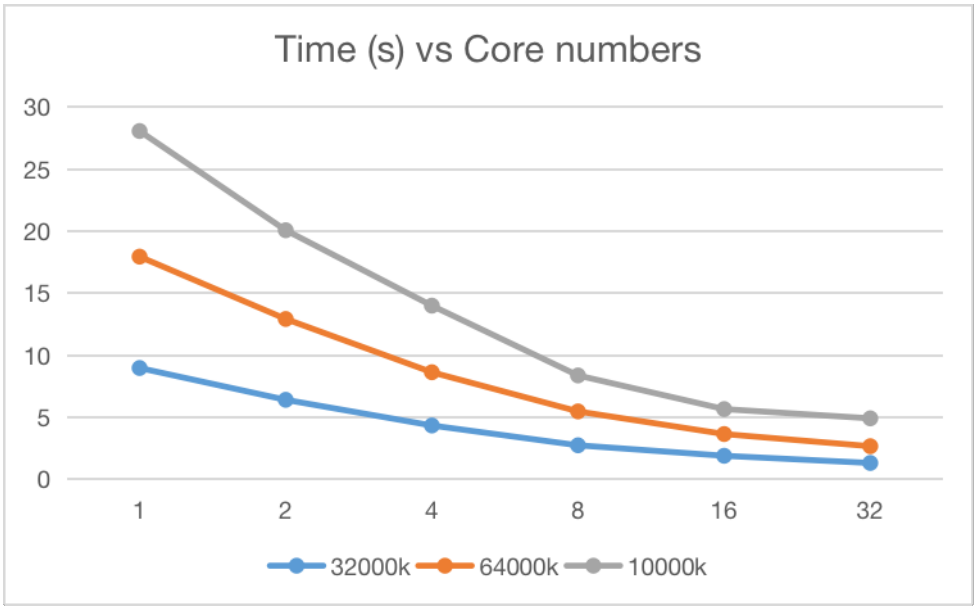


Figure 10: Time vs Core number (big size)

Pthread	1	2	4	8	16	32
360k	0.108878	0.083169	0.062426	0.043598	0.032263	0.022892
1000k	0.275174	0.19995	0.13961	0.088217	0.062857	0.051795
4000k	1.066666	0.749948	0.521705	0.317189	0.209289	0.131795
32000k	8.407762	5.874294	4.004857	2.343809	1.309175	0.815809
64000k	16.817923	11.731293	8.005142	4.660644	2.594556	1.603424
10000k	26.272907	18.331921	12.470767	7.295072	4.037791	2.585286

Table 3: Runtime of pthread

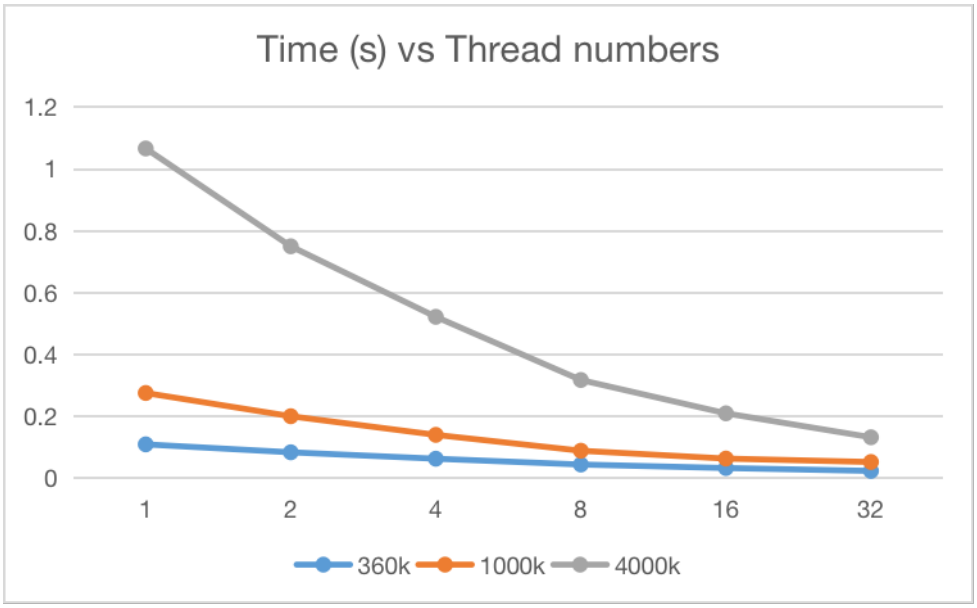


Figure 11: Time vs Thread number (small size)

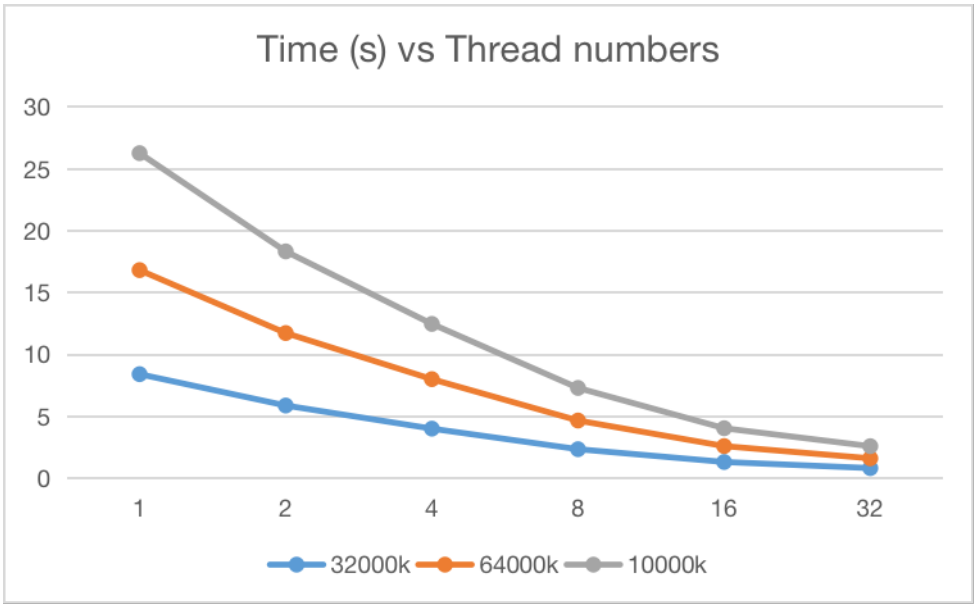


Figure 12: Time vs Thread number (big size)

3.3.4 Speedup

The speedup is given as:

$$Speedup = \frac{\text{running time on one server}}{\text{running time on parallel server}}$$

Figure 13-14 and Table 4 show that speedup increases with the increment of core (thread) number. That is because smaller sub-task size takes less runtime. Besides, larger problem size tends to have a more obvious increment on speedup with the growth of the process (thread) number. The reason may be that the ratio of calculation time over communication time is larger for the sub-task size is large enough.

3.3.5 Efficiency

Efficiency is given as

$$Efficiency = \frac{\text{Speedup}}{\text{process number}}$$

MPI	1	2	4	8	16	32
360k	1	1.246324776	1.682387432	2.773072626	3.203278265	4.154155159
1000k	1	1.373932571	1.941707642	2.999307533	4.317968194	4.945878344
4000k	1	1.397199341	2.023379285	3.214864454	4.951833612	6.326519004
32000k	1	1.401227327	2.071809811	3.284914387	4.76193406	6.885056217
64000k	1	1.389272122	2.081726304	3.286144403	4.936073942	6.762920548
10000k	1	1.398595504	2.006637924	3.357878948	4.971668267	5.729131845

Table 4: Speedup of MPI

Pthread	1	2	4	8	16	32
360k	1	1.30911758	1.74411303	2.497316391	3.374701671	4.756159357
1000k	1	1.376214054	1.971019268	3.119285399	4.377778131	5.312752196
4000k	1	1.422319948	2.044576916	3.362871979	5.096617596	8.093372283
32000k	1	1.431280423	2.099391314	3.587221484	6.422183436	10.30604222
64000k	1	1.433595001	2.100890028	3.608497667	6.482004243	10.48875594
10000k	1	1.433178061	2.106759512	3.601459588	6.506752578	10.16247603

Table 5: Speedup of pthread

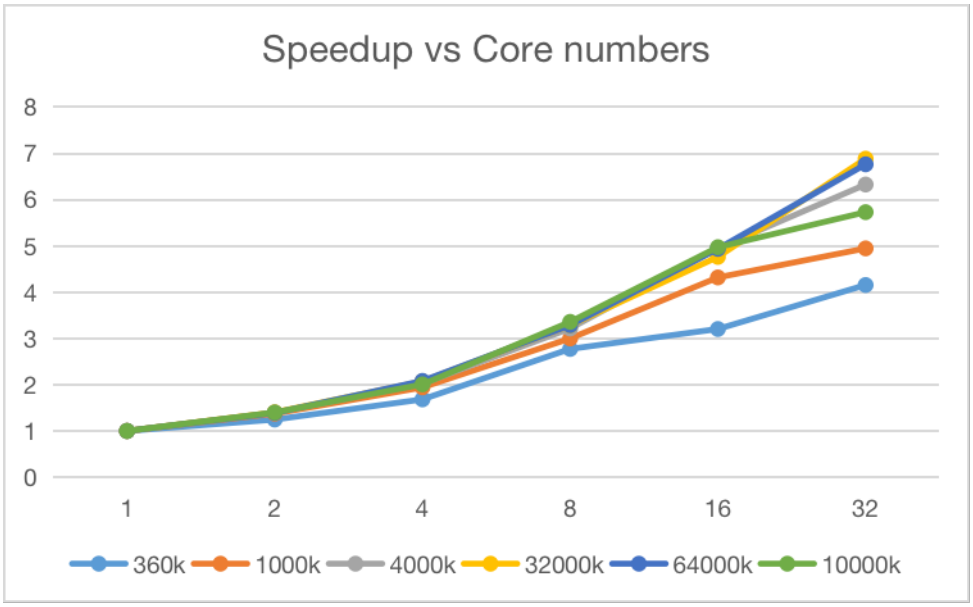


Figure 13: Speedup based on different input size and core number

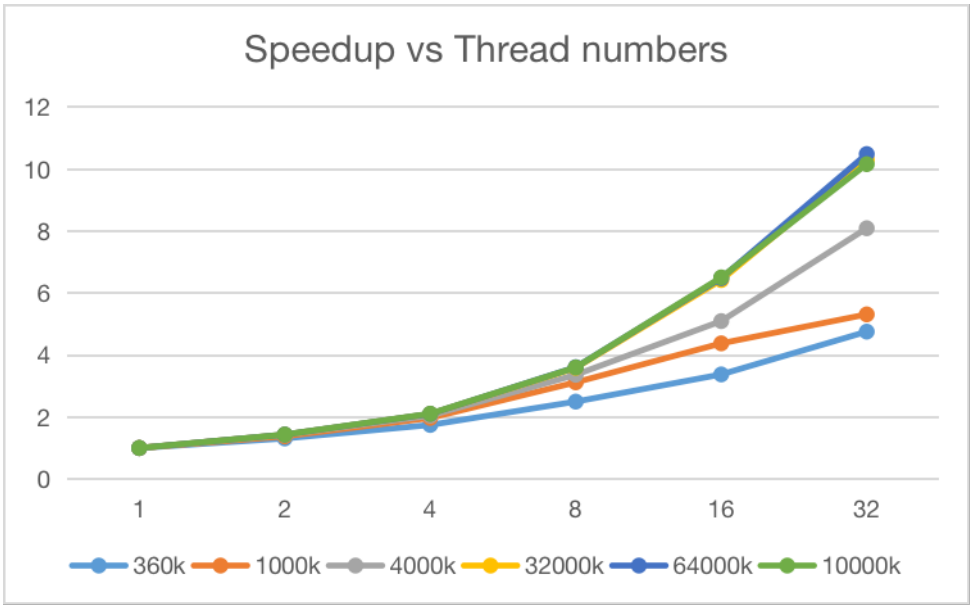


Figure 14: Speedup based on different input size and thread number

Figure 15 and Table 6 show that efficiency drops with the increasing of process number for all data size. One possible reason is that when process number is larger, the data size distributed to sub-process decreases, so the real computing time proportion drops. Meanwhile, more time is used for communication overhead, e.g. waiting for passing data.

MPI	1	2	4	8	16	32
360k	1	0.623162388	0.420596858	0.346634078	0.200204892	0.129817349
1000k	1	0.686966285	0.48542691	0.374913442	0.269873012	0.154558698
4000k	1	0.69859967	0.505844821	0.401858057	0.309489601	0.197703719
32000k	1	0.700613663	0.517952453	0.410614298	0.297620879	0.215158007
64000k	1	0.694636061	0.520431576	0.41076805	0.308504621	0.211341267
10000k	1	0.699297752	0.501659481	0.419734868	0.310729267	0.17903537

Table 6: Efficiency of MPI

Pthread	1	2	4	8	16	32
360k	1	0.65455879	0.436028257	0.312164549	0.210918854	0.14862998
1000k	1	0.688107027	0.492754817	0.389910675	0.273611133	0.166023506
4000k	1	0.711159974	0.511144229	0.420358997	0.3185386	0.252917884
32000k	1	0.715640211	0.524847829	0.448402686	0.401386465	0.322063819
64000k	1	0.7167975	0.525222507	0.451062208	0.405125265	0.327773623
10000k	1	0.716589031	0.526689878	0.450182449	0.406672036	0.317577376

Table 7: Efficiency of pthread

Figure 16 and Table 7 show that efficiency drops with the increasing of thread number for all data size. One possible reason is that the data size distributed to sub-process decreases, so the real computing time proportion drops. Meanwhile,

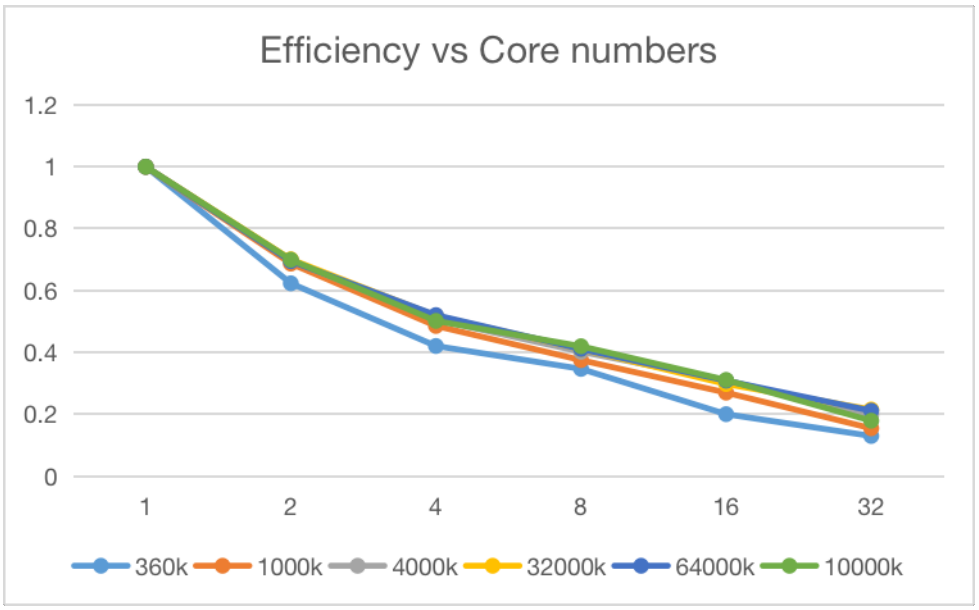


Figure 15: Efficiency based on different input size and core number

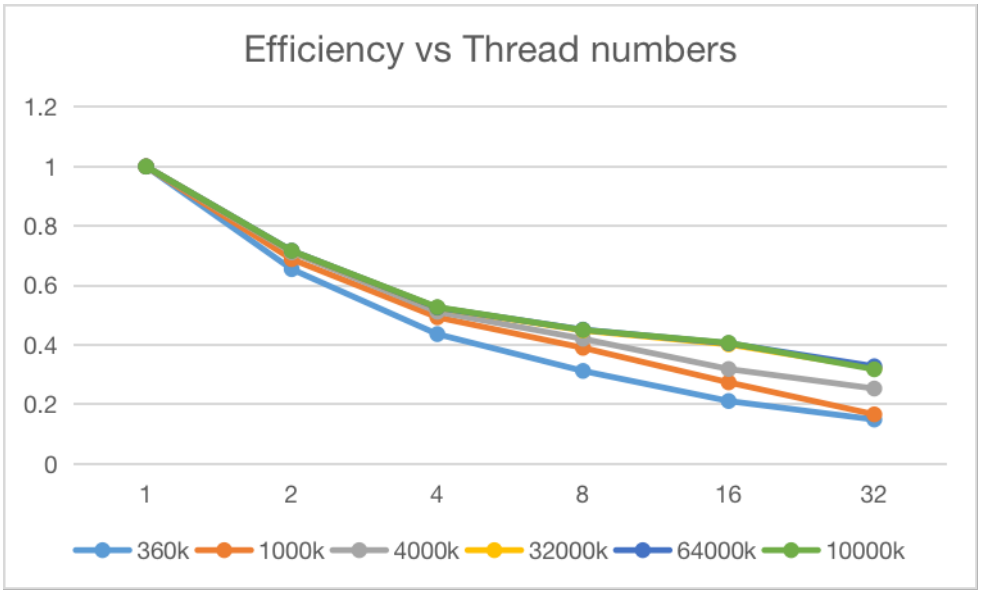


Figure 16: Efficiency based on different input size and thread number

more time is used for thread switching.

4 Conclusion

This report discuss the implementation of MPI and Pthread version of Mandelbrot set computation. It compares runtime sequential, MPI and Pthread. It also analyses the performance of MPI and Pthread version based on different problem sizes and core (thread number) by present graphs and tables of runtime, speedup and efficiency.

For further improvement, we can modify the compute function. In this program, we assume time complexity for one point is $O(M)$, where M is max_iteration. However, the real situation also depends on the required squares. Therefore, we can divide execution into equal time-consuming task rather than equal data size task (dynamic version), which can save waiting time.

5 Source code

5.1 Pthread version

```
1 #include "asg2.h"
2 #include <stdio.h>
3 #include <pthread.h>
4
5 int n_thd; // number of threads
6
7 typedef struct {
8     int a;
```

```
9       int b;
10  } Args;
11
12
13  void* worker(void* args) {
14      //TODO: procedure in each threads
15      // the code following is not a necessary, you can
            replace it.
16
17      Args* my_arg = (Args*) args;
18      int a = my_arg->a;
19      int b = my_arg->b;
20      int block = (total_size + b - 1) / b;
21      int upper = (a+1)*block < total_size ? (a+1)*block
            : total_size;
22      // printf("%d %d %d\n",a,b,block);
23      Point* p = data;
24      for(int idx = a * block; idx < upper; idx++){
25          int k = compute(p + idx);
26          //printf("%d %d %f\n", idx, k, (p+idx)->color);
27      }
28      //TODO END
29      pthread_exit(NULL);
30
31  }
32
33
```

```
34 int main(int argc, char *argv[]) {
35
36     if ( argc == 5 ) {
37         X_RESN = atoi(argv[1]);
38         Y_RESN = atoi(argv[2]);
39         max_iteration = atoi(argv[3]);
40         n_thd = atoi(argv[4]);
41     } else {
42         X_RESN = 1000;
43         Y_RESN = 1000;
44         max_iteration = 100;
45         n_thd = 4;
46     }
47
48     #ifdef GUI
49     glutInit(&argc, argv);
50     glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
51     glutInitWindowSize(500, 500);
52     glutInitWindowPosition(0, 0);
53     glutCreateWindow("Pthread");
54     glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
55     glMatrixMode(GL_PROJECTION);
56     gluOrtho2D(0, X_RESN, 0, Y_RESN);
57     glutDisplayFunc(plot);
58     #endif
59
60     /* computation part begin */
```

```
61      t1 = std::chrono::high_resolution_clock::now();
62
63      initData();
64      //TODO: assign jobs
65      pthread_t thds[n_thd]; // thread pool
66      Args args[n_thd]; // arguments for all threads
67      for (int thd = 0; thd < n_thd; thd++){
68          args[thd].a = thd;
69          args[thd].b = n_thd;
70      }
71       for (int thd = 0; thd < n_thd; thd++)
            pthread_create(&thds[thd], NULL, worker, &args[
            thd]);
72       for (int thd = 0; thd < n_thd; thd++) pthread_join
            (thds[thd], NULL);
73      //TODO END
74      t2 = std::chrono::high_resolution_clock::now();
75      time_span = t2 - t1;
76      /* computation part end */
77
78      printf("Student ID: 120090266\n"); // replace it
           with your student id
79      printf("Name: Feng Yutong\n"); // replace it with
           your name
80      printf("Assignment 2 Pthread\n");
81      printf("Run Time: %f seconds\n", time_span.count())
           ;
```

```
82      printf("Problem Size: %d * %d, %d\n", X_RESN,
            Y_RESN, max_iteration);
83      printf("Thread Number: %d\n", n_thd);
84
85
86
87      #ifdef GUI
88      glutMainLoop();
89      #endif
90      pthread_exit(NULL);
91      return 0;
92  }
```

5.2 MPI version

```
1  #include "asg2.h"
2  #include <stdio.h>
3  #include <mpi.h>
4
5
6  int rank;
7  int world_size;
8
9
10 int main(int argc, char *argv[]) {
11          if ( argc == 4 ) {
12                  X_RESN = atoi(argv[1]);
```

```
13                Y_RESN = atoi(argv[2]);
14                max_iteration = atoi(argv[3]);
15        } else {
16                X_RESN = 1000;
17                Y_RESN = 1000;
18                max_iteration = 100;
19        }
20
21        if (rank == 0) {
22                #ifdef GUI
23                glutInit(&argc, argv);
24                glutInitDisplayMode(GLUT_SINGLE |
                      GLUT_RGB);
25                glutInitWindowSize(500, 500);
26                glutInitWindowPosition(0, 0);
27                glutCreateWindow("MPI");
28                glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
29                glMatrixMode(GL_PROJECTION);
30                gluOrtho2D(0, X_RESN, 0, Y_RESN);
31                glutDisplayFunc(plot);
32                #endif
33        }
34
35        /* computation part begin */
36        struct pointtype b;
37        MPI_Datatype myvar;
38        MPI_Datatype old_types[2];
```

```
39          MPI_Aint indices[2];
40          int blocklens[2];
41
42          MPI_Init(&argc, &argv);
43
44          blocklens[0] = 2;
45          blocklens[1] = 1;
46          old_types[0] = MPI_INT;
47          old_types[1] = MPI_FLOAT;
48          MPI_Address(&b, &indices[0]);
49          MPI_Address(&b.color, &indices[1]);
50          indices[1] -= indices[0];
51          indices[0] = 0;
52          MPI_Type_struct(2,blocklens,indices,old_types,&
                myvar);
53          MPI_Type_commit(&myvar);
54
55          MPI_Comm_rank(MPI_COMM_WORLD, &rank);
56      MPI_Comm_size(MPI_COMM_WORLD, &world_size);
57
58
59          if (rank == 0) {
60                  t1 = std::chrono::high_resolution_clock
                        ::now();
61                  initData();
62          }
```

```
63          MPI_Bcast(&total_size, 1, MPI_INT, 0,
                MPI_COMM_WORLD);
64          int block = total_size / world_size;
65          //printf("%d %d %d\n",block, total_size,
                world_size);
66          Point*  mydata;
67          mydata = new Point[block];
68          MPI_Scatter(data, block, myvar, mydata, block,
                myvar, 0, MPI_COMM_WORLD); // distribute
                elements to each process
69
70          MPI_Gather(mydata, block, myvar, data, block,
                myvar, 0, MPI_COMM_WORLD);
71
72          MPI_Finalize();
73          /* computation part end */
74
75          if (rank == 0){
76
77                  t2 = std::chrono::high_resolution_clock
                        ::now();
78                  time_span = t2 - t1;
79
80                  printf("Student ID: 120090266\n"); //
                        replace it with your student id
81                  printf("Name: FengYutong\n"); //
                        replace it with your name
```

```
82                  printf("Assignment 2 MPI\n");
83                  printf("Run Time: %f seconds\n",
                        time_span.count());
84                  printf("Problem Size: %d * %d, %d\n",
                        X_RESN, Y_RESN, max_iteration);
85                  printf("Process Number: %d\n",
                        world_size);
86
87
88                  #ifdef GUI
89                  glutMainLoop();
90                  #endif
91          }
92
93          return 0;
94  }
```