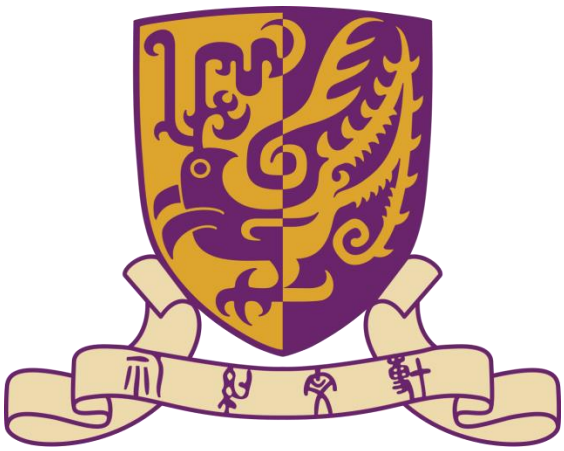


THE CNINESE UNIVERSITY OF HONG KONG, SHENZHEN

CSC4005

Distributed and Parallel Computing

Report for Assignment 2
Mandelbrot Set Computation

Author:
Li Jingyu 李璟瑜

Student Number:
118010141

Nov 1, 2021

Contents

1 Introduction

Assignment 2 requires us to design a program for Mandelbrot set computation and render the points on screen with a type of GUI. In a complex plane, a number is represented by its real part (x-axis) as well as the image part (y-axis) [z = a + b*i*]. By performing some kind of transformation, the point is mapped to another point in the plane, forming a special shape. Mandelbrot set computation is basically illustrated by these formulas:

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

$$z^2 = a^2 + 2abi + bi^2 = a^2 - b^2 + 2abi$$

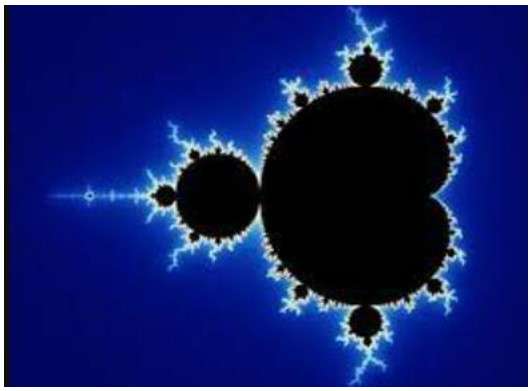
$$z_{\text{real}} = z_{\text{real}}^2 - z_{\text{imag}}^2 + c_{\text{real}} \quad z_{\text{imag}} = 2z_{\text{real}}\, z_{\text{imag}} + c_{\text{imag}}$$

Here *z* is a complex number in the plane and k represents the iteration number. The parameter *c* here is calculated by:

$$c = \frac{x - \text{height}/2}{\text{height}/4} + \frac{y - \text{width}/2}{\text{width}/4} \times i.$$

Scaling c with different calculation, the generated shape will change accordingly.

By updating the position of different points in the complex plane, some points will go beyond certain range, running to infinity, while the other points will remain stable, rising or dropping in a specific area, which are called “quasi-stable”. Our task in the assignment is to render those “quasi-stable” points fluctuating with limit in the screen using ImGui (a C++ supported graphical user interface) and explore the relationship between computation speed and different parameters.



The sequential version code is given while we should attempt to implement the parallel versions using **MPI** and **Pthread** approaches. MPI (Massage Passing Interface) is the standard of message passing function library that includes many interfaces. It is a parallel technology based on the communication between processes. A sample MPI interface is shown as follows. MPI_Send is one of the most popular functions in MPI.

```
MPI_Send(
    void* data,
    int count,
    MPI_Datatype datatype,
    int destination,
    int tag,
    MPI_Comm communicator);
```

```
int pthread_create(
    pthread_t *restrict thread,
    const pthread_attr_t *restrict attr,
    void *(*start_routine)(void *),
    void *restrict arg);
```

Pthread refers to POSIX threads, which is an execution model that exists independently from a language and a parallel execution model. It is primitive and practical in C programming. Here shows the basic grammar of creating a thread and synchronizing the thread.

```
int pthread_join(pthread_t thread, void** return_value);
```

The core part of this assignment is to partition the image reasonably and allocate the pixels for computation, and then synchronize the process/thread and aggregate the data for image rendering.

2 Design

In this part the logic of the codes and some details will be elaborated. Both the MPI and Pthread versions are based on the ImGui template on blackboard.

- MPI version

In MPI programming, the programmer should attach importance to the message passing between different processes.

In the main function, initialize the MPI setting first. Then create two local buffers (int pointer) storing the local computation result. Also create the global buffer in each process as the parameter of the calculation function.

In the master process, set up the graphical context for plotting. The rendering function will run repeatedly in order to obtain more average and accurate speed. The variables related to Mandelbrot set computation is initialized (e.g. size of window = 800). Then Set some dragger for adjusting the parameters dynamically in the run time. (However, the procedure is usually slow. So it would better fix the parameters in advance) Resize the global buffer to a *size* * *size* square, then start computation part.

The following part will be treated as the computation cost (time). We only care about computation time in this project, since drawing costs hugely in the GUI rendering context that is often done with a single process/thread.

In the calculation, the related parameters going to be passed to the functions are broadcast to all the processes in MPI_COMM_WORLD. Then allocate the space for two local buffers. Enter the function. **In the computation part, each row of the**

square is evenly distributed to each process, one row for one process. In this way, the workload can be balanced better than allocating the rows using "MPI_Scatter" function directly, since **some rows with "quasi-stable" points require more computation, and these rows are usually continuous**. By allocating the row cross wisely, the workload is more balanced. The calculation function part goes according to the formula, but the different part is about the remaining lines. Because in some scenarios, t**he size (the number of rows) is not divisible by the process number**, so it is necessary to **deal with the remaining rows**. This explains the usage of the second local buffer "remain" in the code. For the rows left out after distributing tasks to each process evenly, the calculation result will be stored in another buffer passed back to the corresponding process. The function is shown as follows.

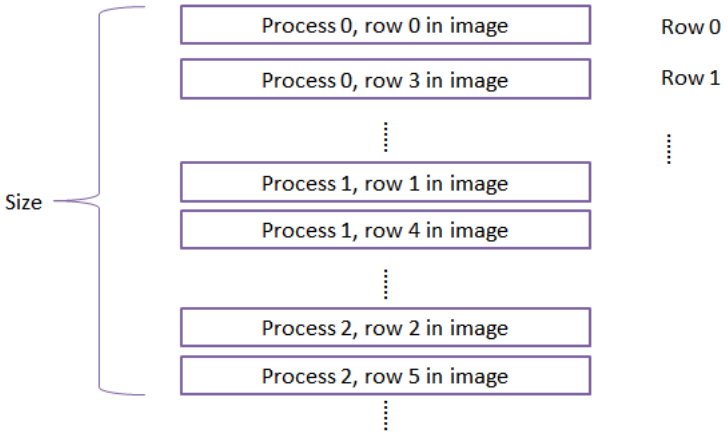
```
void calculate(int* local, int* remain, int rank, int proc_num, int size, double scale, double x_center, double y_center, int k_value) {

    double cx = static_cast<double>(size) / 2 + x_center;
    double cy = static_cast<double>(size) / 2 + y_center;
    double zoom_factor = static_cast<double>(size) / 4 * scale;
    int base = 0;

    for (int i = rank; i < size; i += proc_num) {  // distribute row by row
        for (int j = 0; j < size; j++) {
            double x = (static_cast<double>(j) - cx) / zoom_factor;
            double y = (static_cast<double>(i) - cy) / zoom_factor;
            std::complex<double> z{0, 0};
            std::complex<double> c{x, y};
            int k = 0;
            do {
                z = z * z + c;
                k++;
            } while (norm(z) < 2.0 && k < k_value);  // only paint the points iterting k_value times
            if (i < size / proc_num * proc_num) {
                local[base + j] = k;
            }
            else {
                remain[j] = k;
            }
        }
        base += size;
    }
```

After performing the function, use "*MPI_Gather*" to gather the all results in different processes. Then deal with the remaining rows, receive the line data from the processes that have conducted the extra work. Free the buffers afterwards.

The calculation part then finishes.

The computation speed will be calculated, whose value equals to the processed pixels (size, length of square) divided by the duration, which is written in template.

Because I distribute the tasks **line by line** in parallel function, the returning global buffer is not exactly in the same order as the real image. The global buffer ordering should be like this (Assume there are 3 processes):



So a row mapping is required to plot the graph correctly.

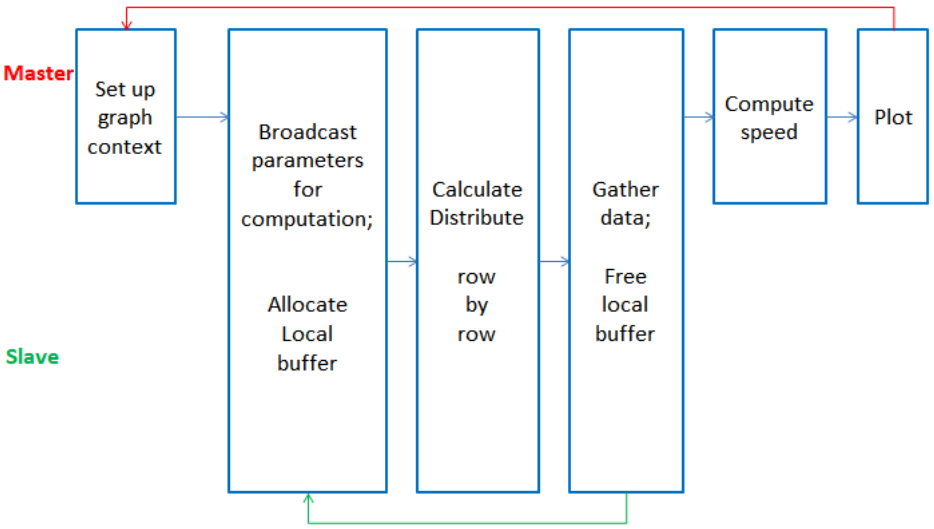
```
// Because the point allocation way is different, the drawing procedure is requi
for (int i = 0; i < size; i++) {
    int i_new;
    if (i < size / proc_num * proc_num) {
        i_new = i % proc_num * (size / proc_num) + i / proc_num;
    } else {  // the remaining part is in order
        i_new = i;
    }

    for (int j = 0; j < size; j++) {
        if (canvas[{i_new, j}] == k_value) {
            draw_list->AddCircleFilled(ImVec2(x, y), radius, col32);
            // std::cout << i << " " << j << std::endl;
        }
        x += spacing;
    }
    y += spacing;
    x = p.x + MARGIN;
}
```

The master process will continue to assign the parameter value and check whether there is a change and run the computation function, calculate time cost and plot the graph.

For the slave processes, they will receive the parameters broadcast by master process, allocate the two local buffers and then start calculation with these values. After computation, return the data to master process and send the remaining row. The salve processes are only responsible for computation but not plotting, and they will perform repeatedly.

Computation part in master process and a structural flow figure are displayed.

```
auto begin = high_resolution_clock::now();
MPI_Bcast(&center_x, 1, MPI_INT, 0, MPI_COMM_WORLD);  // Broadcast all the meta parameters
MPI_Bcast(&center_y, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&size, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&scale, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&k_value, 1, MPI_INT, 0, MPI_COMM_WORLD);

local = (int*)malloc(size * (size / proc_num) * sizeof(int));  // allocate local buffer
remain = (int*)malloc(size * sizeof(int));

calculate(local, remain, rank, proc_num, size, scale, center_x, center_y, k_value);
MPI_Gather(local, size * (size / proc_num), MPI_INT, canvas.pointer(), size * (size / proc_num), MPI_

// Copy the remaining line in master process
if (size % proc_num != 0) {
    for (int i = 0; i < size; i++) {
        *(canvas.pointer() + size * (size / proc_num) * proc_num + i) = *(remain + i);
    }
}

// Copy the remaining line in slave process
for (int i = 1; i < size % proc_num; i++) {
    MPI_Recv(canvas.pointer() + size * ((size / proc_num) * proc_num + i), size, MPI_INT, i, MASTER,
}

free(local);
free(remain);
auto end = high_resolution_clock::now();
```



- Pthread version

In Pthread programming, the programmer should lay more emphasis on thread synchronization and avoid dead lock.

Compared with MPI version, Pthread is similar but easier to implement. This time, no local buffer is required and all the threads will operate the global buffer directly since they share the same memory space. The parameters for calculation are initialized as global variables, too. To represent different thread, I use an atomic integer as the thread number (rank in MPI version), which can avoid deadlock, ensure thread safety and provide better performance than *mutex*.

```
std::atomic_int current_thread{0};
```

std::atomic

Defined in header <atomic>

```
template< class T >
struct atomic;                                  (1)  (since C++11)

template< class U >
struct atomic<U*>;                              (2)  (since C++11)
```

Defined in header <memory>

```
template< class U >
struct atomic<std::shared_ptr<U>>;              (3)  (since C++20)

template< class U >
struct atomic<std::weak_ptr<U>>;                (4)  (since C++20)
```

Defined in header <stdatomic.h>

```
#define _Atomic(T) /* see below */              (5)  (since C++23)
```

Figure: Atomic Classes

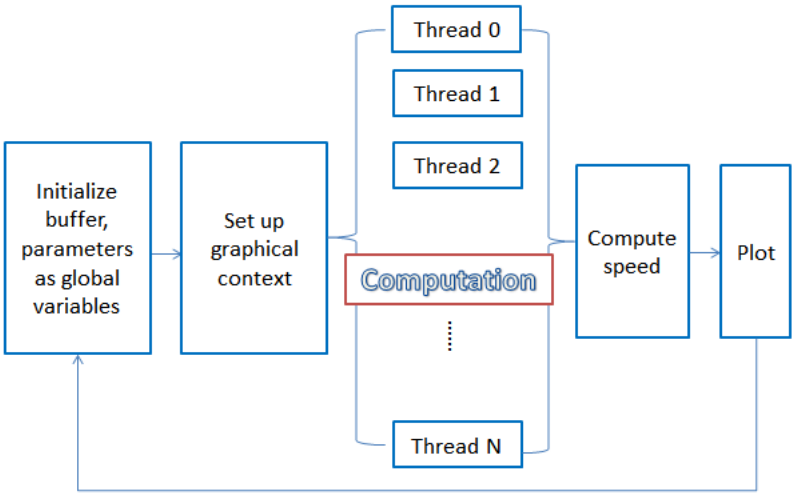
In the main function, the program receives thread number from command line. Similarly, the main thread will set up graphical context. In the calculation part, create threads and execute the calculation function. This time, no operation for remaining part is needed and each thread *i* just deal with the rows whose remainder after dividing the *size* equals to *i*. After iteration of the point, modify the specific position in global buffer directly. The funciton shows as follows.

```
void *calculate(void *) {
    int rank = current_thread;
    current_thread++;

    double cx = static_cast<double>(size) / 2 + center_x;
    double cy = static_cast<double>(size) / 2 + center_y;
    double zoom_factor = static_cast<double>(size) / 4 * scale;

    for (int i = rank; i < size; i += thread_num) {  // distribute row by row
        for (int j = 0; j < size; j++) {
            double x = (static_cast<double>(j) - cx) / zoom_factor;
            double y = (static_cast<double>(i) - cy) / zoom_factor;
            std::complex<double> z{0, 0};
            std::complex<double> c{x, y};
            int k = 0;
            do {
                z = z * z + c;
                k++;
            } while (norm(z) < 2.0 && k < k_value);  // only paint the points itert
            canvas[{i, j}] = k;
        }
    }

    pthread_exit(NULL);
}
```

After invoking the function, join all the threads. Compute speed and plot the graph, then start next repetition. The whole process is relatively simple.

Computation part and program flow show as follows.

```
/* Start calculation */
auto begin = high_resolution_clock::now();
current_thread = 0;
for (int i = 0; i < thread_num; i++) {
    pthread_create(&cal_thread[i], nullptr, calculate, nullptr);
}
for (int i = 0; i < thread_num; i++) {
    pthread_join(cal_thread[i], nullptr);
}

auto end = high_resolution_clock::now();
/* Finish calculation */
```



- Some details

Only the computation part makes sense when calculating the speed, since plotting costs greatly by a single thread.

The program runs repeatedly to average the speed and obtain more accurate results.

The method I use is static allocation, partitioning the square as rows and distributing row by row, which balances the workload to some degrees. If there is a remaining row, just send one extra row data.

I have changed the scale to *double* type and set the value to 0.5 to view the whole graph. The scale is fixed since changing the scale will make the code stuck.

It is meaningless to pass pointer in MPI programming since in different machine the memory address is not the same.

In MPI, a process cannot receive the message from itself [bug].

I do not set the *size* or *k_value* as **command line input** since they can be adjusted in the interactive GUI window.

The default image size is 800 * 800.

3 Execution

The execution for my code performs on the remote cluster. There are 8 internal nodes each with one **NVIDIA 2080Ti GPU card** with 32 cores together. The maximum CPU time limit is 640 (4 nodes, 128 cores for 5 minutes).

Environment: Linux operated with MPI library and Pthread.

Way to compile and execute:

1. Unzip 118010141.zip
2. cd csc4005-assignment-2 & csc4005-imgui-mpi
3. mkdir build && cd build
4. cmake .. -DCMAKE_BUILD_TYPE=Release
5. cmake --build . -j4

```
cd /path/to/project
mkdir build && cd build
cmake .. -DCMAKE_BUILD_TYPE=Debug # please modify this to `Release` if you
want to benchmark your program
cmake --build . -j4
```

6. **mpirun -np 4 ./csc4005_imgui** [Can specify the process number here]
7. Adjust the Center X & Y as negative value to view the image (drag left)
8. Check the speed in the bash.
9. **Ctrl+C** to **forcedly** end the program.

10. [return] cd csc4005-imgui-pthread
11. mkdir build && cd build
12. cmake .. -DCMAKE_BUILD_TYPE=Release

13. cmake --build . -j4
14. **./csc4005_imgui 4** [Can specify the thread number here]
15. Adjust the Center X & Y as negative value to view the image (drag left)
16. Check the speed in the bash.
17. **Close** the window to stop the program

Some notice:

For graphical exception:

Use ssh –Y to register.

In /pvfsmnt/118010141 directory:

I run in **/pvfsmnt/118010141**, the share file system.

For all the experiment, I use ***salloc*** to run interactively:

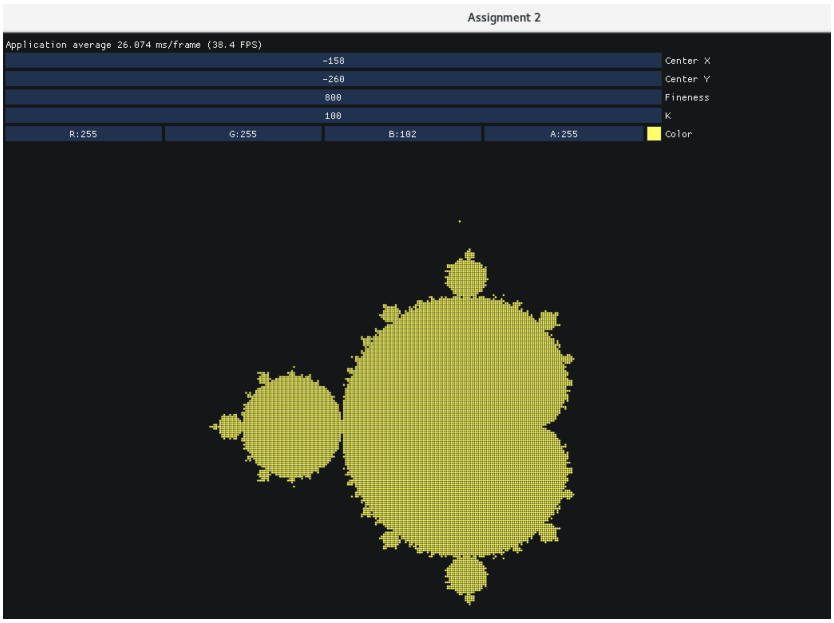
Salloc –N1 –n32 –t10

Again, the program should exit after a certain time limit defined by the *sbatch* script or forcedly ended by the user [ctrl + C] since it is running repeatedly.

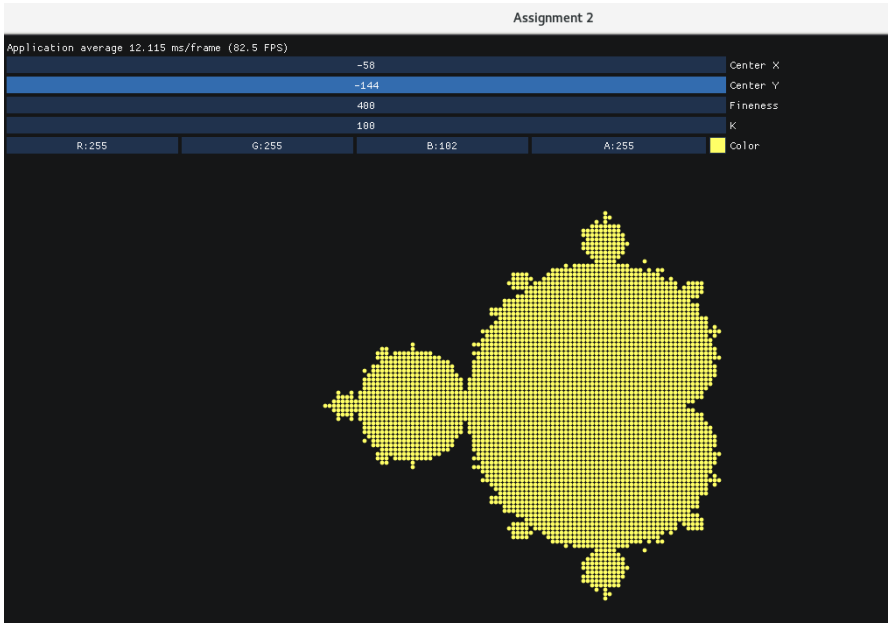
4 Result & Analysis

- GUI Result

The GUI window is shown as follows. To view the complete graph, drag the X center and the Y center accordingly.

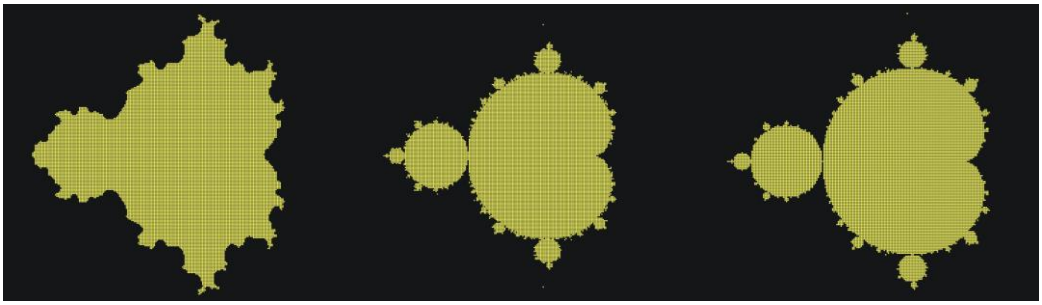


Adjust the size to obtain a graph of different fineness. The larger the size, the finer the graph is. If the size is small, the grain should be rough (easier to see the small circle) [size = 400]



Adjust the k_value to obtain a slightly different image in the margin. The larger the k, the fewer points there are [but the reduction is small], the longer it will cost to compute. The smaller the k_value, the more points included.

[k_value = 10, 100, 1000]



- Bash Result

In the bash, the computing time (close to 0.5 second) as well as the pixels processed in this time period is shown. The speed is calculated then. The output is continuous.

```
[csc4005@localhost build]$ mpirun -np 4 ./csc4005_imgui
[WARN] cannot get screen dpi, auto-scaling disabled
32800 pixels in last 503778721 nanoseconds
speed: 65108 pixels per second
43200 pixels in last 510295073 nanoseconds
speed: 84656.9 pixels per second
40000 pixels in last 502578802 nanoseconds
speed: 79589.5 pixels per second
43200 pixels in last 546683951 nanoseconds
speed: 79021.9 pixels per second
40000 pixels in last 505844473 nanoseconds
speed: 79075.7 pixels per second
^C[csc4005@localhost build]$
```

```
[csc4005@localhost build]$ ./csc4005_imgui 10
[WARN] cannot get screen dpi, auto-scaling disabled
45600 pixels in last 504448508 nanoseconds
speed: 90395.7 pixels per second
45600 pixels in last 500986366 nanoseconds
speed: 91020.4 pixels per second
Segmentation fault (core dumped)
```

- Robustness

The program is able to deal with the case with different process/thread number (1-N), different size, different k_value, different scale (need to modify the code), different X/Y center position, and so on.

- Relationship between speed & process(thread) number

To explore the relationship between running speed in pixels per second (y-axis) and the process (thread) number (x-axis), the following experiments are presented.

Using the control-variable method, the size (length of square), k_value and scale should be fixed among different process/thread numbers. In the experiment group, I set 3 groups of size, namely small (400), medium (800) and large (1600). MPI version and Pthread version are measured separately. In each group, process/thread number from **1** (sequential) to **10** will be tested (if the number is larger, the graphical window responses pretty slow, so 10 is sufficient to show some insights), whose speed (pixels processed / duration time) will be recorded and graphed in a line chart to make the comparison and analyze the tendency.

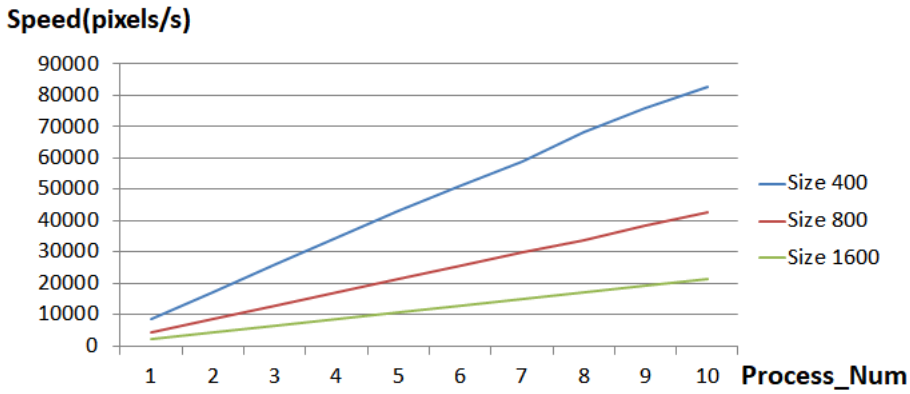
The detailed data and the line chart are shown as follows.

The unit for the value is **pixels/s**, which will not be explained again later.

MPI version:

Process Number	Size 400	Size 800	Size 1600
1	8758.37	4362.6	2195.14
2	17371.4	8730.4	4366.24
3	25955.9	13018.7	6548.11
4	34452.6	17252.1	8702.75
5	43090.3	21433.6	10850.8
6	51370.6	25542.9	12965.8
7	59077.3	29775.5	15075.4
8	68212.8	33954.2	17135.9
9	76138.3	38477.7	19163.2
10	82633.8	42649.8	21338

Relationship between speed (pixels/s) & process number (1-10), MPI version



As can be seen from the raw data and the graph, the program runs the most slowly sequentially, and then the speed increases linearly as process number increases, forming a **perfect straight line**! The slope is fixed in all the three cases, meaning that the average computation speed is fixed among different processes. The result shows the advantage of my approach for distribution——row by row, crossed. The balanced workload renders each process similar time to finish the task and return the results back for plotting. There is little interference issue like process synchronization or data transfer events when the process number is relatively small (1-10).

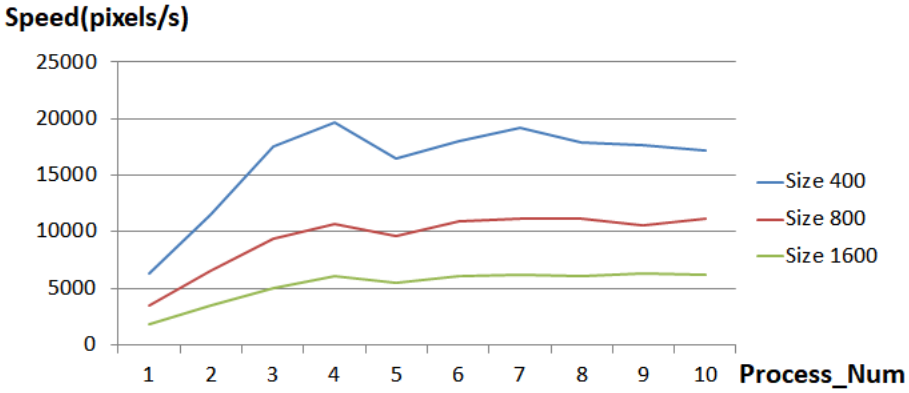
Checking the lines with different colors (size), we can find that the smaller the size is, the greater the speed is. If the size is small, the processing procedure should be fast and thus leads to a larger speed value and a larger slope in the graph. If analyzing quantitatively, the speed forms a reciprocal double ratio, which means when the size doubles, the speed reduces nearly by half. (e.g. **8758.3 / 4342.6 ≈ 2**) This is a wonderful conclusion, proving the processing time is in **linear relationship** with the problem size.

In conclusion, the MPI version performance is pretty stable and fits the expectation when the process number is in range 1 to 10.

Pthread version:

Process Number	Size 400	Size 800	Size 1600
1	6261.75	3473.89	1792.55
2	**11499**	6548.51	3429.54
3	**17568.1**	9403.94	5064.78
4	**19668.9**	10631.4	6132.29
5	**16429.8**	9675.44	5487.9
6	**18004.4**	10880	6091.65
7	**19217**	11188	6231.73
8	**17943**	11174.9	6117.06
9	**17687.1**	10581.7	6279.16
10	**17158.7**	11101.8	6245.36

Relationship between speed (pixels/s) & process number (1-10), Pthread version



Unfortunately, the shape of the line chart for Pthread version is not that perfect compared with MPI version. As the thread number ranges from 1 to 3, the line can still keep straight, forming a linear relationship between the speed and the thread number. The ratio is also perfect, just as the MPI version (double). However, when

and after the thread number exceeds 4, the line becomes fluctuated, and goes horizontally after thread number is larger than 7. The Pthread version code encounters a bottleneck at thread number 4. When the thread number continues to increase, it will not lead to a good performance improvement but a stable one. Besides, the data in the bash are pretty unstable during the test, whose speed values range much greater than MPI version. One of the possible reasons to explain the bottleneck and the instability is the cost for thread creation and synchronization. In the main thread, the other threads are created one by one, causing different start time for computation. One thread can only start after the previous thread is successfully launched. As a result, when the thread number is large, the time interval between the first thread and the last thread should be large. After computation, the function Pthread_join() is used to synchronize all the threads, causing a time delay. Assume the time for creating a thread is fixed, it is normal that when the thread number increases, although the computation time can be shortened, the time difference increases, causing a longer waiting time for thread synchronization and no improvement on the total performance due to the elimination.

```
/* Start calculation */
auto begin = high_resolution_clock::now();
current_thread = 0;
for (int i = 0; i < thread_num; i++) {
    pthread_create(&cal_thread[i], nullptr, calculate, nullptr);
}
for (int i = 0; i < thread_num; i++) {
    pthread_join(cal_thread[i], nullptr);
}

auto end = high_resolution_clock::now();
/* Finish calculation */
```

MPI VS Pthread

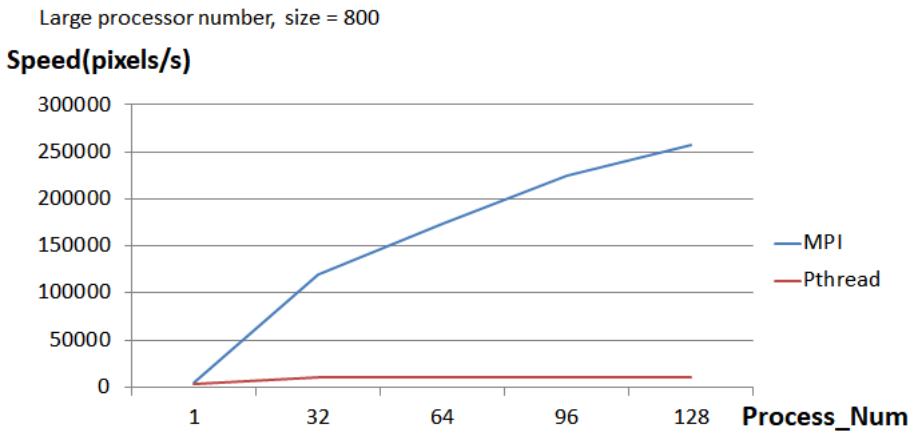
Compared MPI version with Pthread version, firstly, MPI runs faster than Pthread when the process number is the same. This is partly because MPI initializes all the processes at the beginning and this time does not count in the computation part. So the MPI version can start computation just after receiving all the parameters. However, for Pthread, all the threads are created during the computation part, which occupies certain time. The running time is determined by the slowest thread, which waits for other thread to begin its own task. The thread synchronization costs greatly.

Large process number

I made another experiment, setting the number of processes sequential and the multiple of 32. Because every node has 32 cores, I would like to explore the efficacy when the cores are distributed in different GPU nodes for MPI version. To make a comparison, the Pthread version data are also shown.

Process Number	MPI	Pthread
1	4362.6	3473.89
32	120392	11114.8
64	173700	10973.6
96	224888	10466.5
128	257732	11005.5

Relationship between speed (pixels/s) & process number (1-128)



From the raw data and the graph, the running speed increases greatly from 1 process to 32 processes and the **increasing rate** becomes smoother when the process number becomes larger. The speedup is no longer perfectly linear when the core number is relatively huge, although it is still acceptable. Because the cost for process communication and synchronization increases as the number of processes adds up this time. The main cost lies in the data gathering and then the parameters broadcast. Besides, the cost for process communication between different nodes is far greater than that in the same node (32 cores). In the same GPU, the cost is ignorable (proved by my previous experiment), but in the different GPU, the cost cannot be neglected! As a result, structure of the cluster should also be taken into account when analyzing. There are 32 cores in a node, and every one node addition will reduce the speedup ration (the slope in the graph). Also, reasonably, as the square size increases, the reduction of speedup should be less.

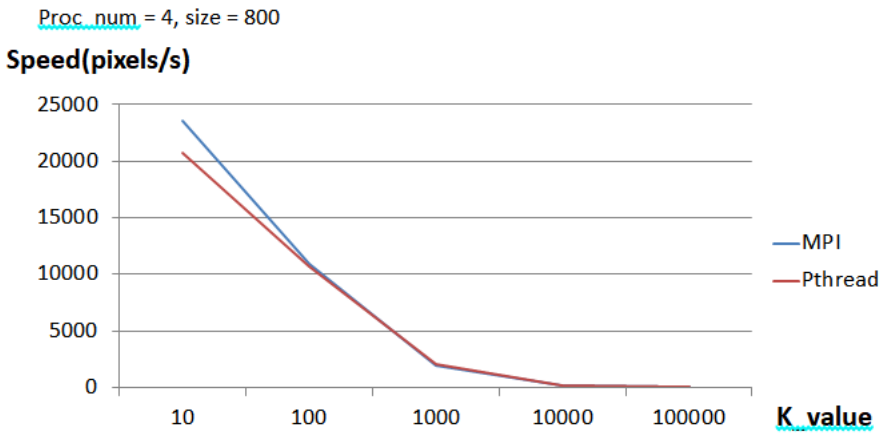
For Pthread version, conforming to my previous analysis, the performance is stable after certain thread number.

- Relationship between speed & k_value

Similar to the previous experiment, to explore the relationship between the speed and k_value, the number of processes (threads) and the size should be fixed. Here I choose the parallel scenario when N=4 and size=800 and fix the values. The k_value varies from 10 to 100000, with 10 times each time. The results are shown as follows.

k_value	MPI	Pthread
10	**23623**	20772.5
100	**10899.9**	10631.4
1000	1902.73	2084.02
10000	**206.944**	211.87
100000	**17.0684**	22.1393

Relationship between speed (pixels/s) & k_value (1e1-1e5)



As can be seen from the raw data and the graph, the speed decreases obviously when k_value multiplies by 10 each time. Starting from k=100, each time k increases by 10 times, the speed also divided by nearly 10 times, fitting our previous conclusion that running time increases nearly linearly with the problem size, and the ratio is fixed. Besides, MPI version result is similar with that of Pthread result. In the code, k_value

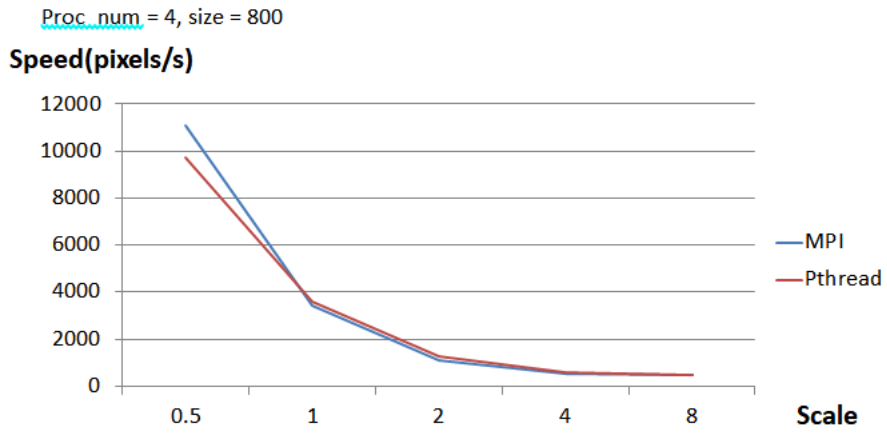
represents the maximum iteration time of transformation, limiting the quasi-stable points. For those points that will go beyond the limit, this value is useless. As a result, the larger the k_value, the more computation resources needed for those quasi-stable points, and thus a smaller speed. Because the number of quasi-stable points is nearly fixed under different k, it will reveal linear relationship between *1/speed* and k_value.

- Relationship between speed & scale

Similar to the previous experiment, to explore the relationship between the speed and scale, I fixed the N=4 and size=800 and k_value=100. The scale varies from 0.5 to 8, doubling each time. The results are shown as follows.

scale	MPI	Pthread
0.5	11106.7	9748.98
1	3434.29	3591.27
2	1110.16	1249.88
4	510.074	581.795
8	487.098	500.561

Relationship between speed (pixels/s) & k_value (1e1-1e5)



As can be seen from the raw data and the graph, the speed reduces when scale doubles each time. There is no obvious rule between the values. MPI version result is closely the same as that of Pthread's result. In the code, scale is related to the zoom_factor. The larger the scale, the larger the zoom factor, the smaller the absolute value of X and Y in the complex plane, and thus it is more difficult to go beyond the norm limit (2) after certain computation (100 iterations). As a result, more computation resources are required, and the speed is lower.

5 Summary

Here I will summarize what I have learned when writing assignment 2.

- Understanding of parallel computing. For example, different process shares different memory, and different threads can share the same global variables. The same pointer cannot always make effect
- The problems to be noticed when writing parallel programs (process synchronization, process communication, thread creation and join, cost)
- MPI & Pthread library usage
- Coding ability improvement
- Analysis ability increases
- Cluster experience. Queuing to obtain the resources! Crowded queue before ddl (especially in the afternoon)
- By the way, tutorials are quite practical for writing the project and TA & USTF are warm-hearted in answering the questions in the wechat Group, Kudos!

That’s all.