Project2: Mandelbrot Set Computation

Name: Xinyu Xie

Student ID: 119020059

Introduction

The Mandelbrot set is the set of complex numbers c for which the function $f_c(z) = z^2 + c$ does not diverge to infinity when iterated from $z = 0$, forming a sequence remaining bounded in absolute value. Images of the Mandelbrot set exhibit an elaborate and infinitely complicated boundary.

In a complex plane, a number is represented by real part and image part, i.e. $Z = a + bi$. Mandelbrot set is a set of points in the complex plane. Set of points are quasistable when computed by iterating the function: $Z_{k+1} = Z_k^2 + c$, where Z_{k+1} is the (k + 1)th iteration of the complex number $Z = a + bi$, Z_k is the kth iteration of Z, and c is a complex number giving the position of the point in the complex plane. The initial value for Z_0 is zero. The iteration are continued until the magnitude of Z_k is greater than some threshold or number of iteration reaches maximum.

The computation of the complex function $Z_{k+1} = Z_k^2 + c$ is simplfied through the following formula:

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

Since the real part is $a^2 - b^2$ and the imaginary part is $2abi$. The next iteration values can be computed by:

$$Z_{\text{real}} = Z_{\text{real}}^2 - Z_{imag}^2 + c_{\text{real}}$$

$$Z_{imag} = 2Z_{\text{real}}\, Z_{\text{imag}} + c_{imag}$$

In this project, I implemented the seqential version, the parallel MPI version, and the parallel Pthread version of Mandelbrot set computation. The detailed implementation methods, results and conclusions are provided in the following sections.

Method

Sequential Mandelbrot Set Computation

The following are the steps to perform sequential Mandelbrot set computation:

1. Create a Point for each pixel in the image, and store all Points in the data array.
2. For each Point in the data array, compute its color using the Mandelbrot set computation formula given in the introduction section.

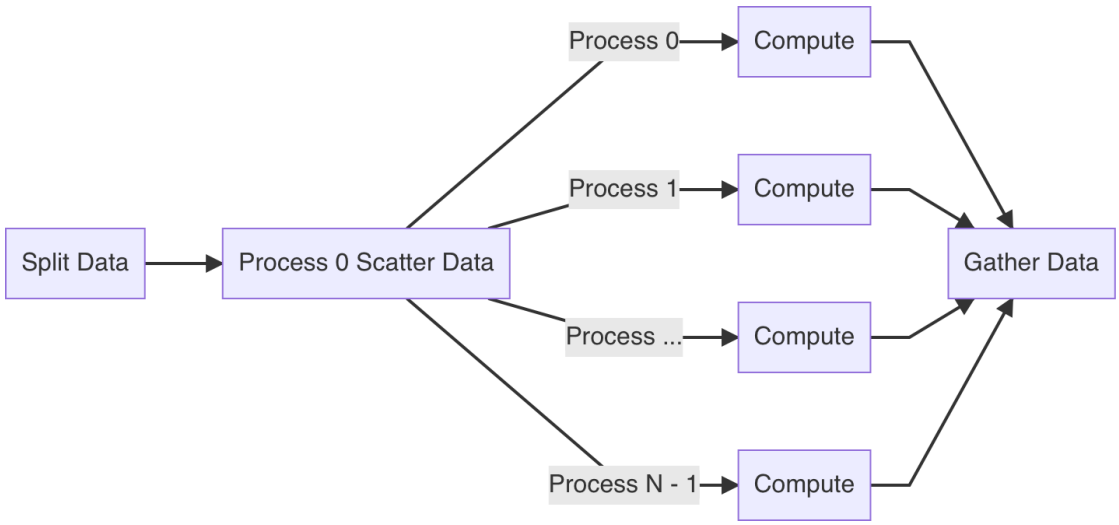
Parallel Mandelbrot Set Computation using MPI

The following are the steps to perform parallel Mandelbrot set computation using MPI:

3. Create a Point for each pixel in the image, and store all Points in the data array.
4. In total, m pixel Points are equally distributed to n processes, respectively. Here, Points are partitioned by columns, the data array storing all data points are partitioned equally to be assigned to different processes.
5. Master process distribute equal number of elements to each slave process.
6. Inside each slave process, for each Point in its local data array, compute the point's color using the Mandelbrot set computation formula given in the introduction section.
7. Master process collect the result from each slave process.

The main idea is as follows. Firstly, total m pixel Points in the image are equally distributed to n processes. The points will be partitioned in the master process. The the master process will distribute the partitioned data points to the slave processes. And in each slave process, the Mandelbrot set computation is performed for each Point in its local data array. Then the master process collect results from all slave process after they finish their computation.

This flow chart is illustrated in the following figure:



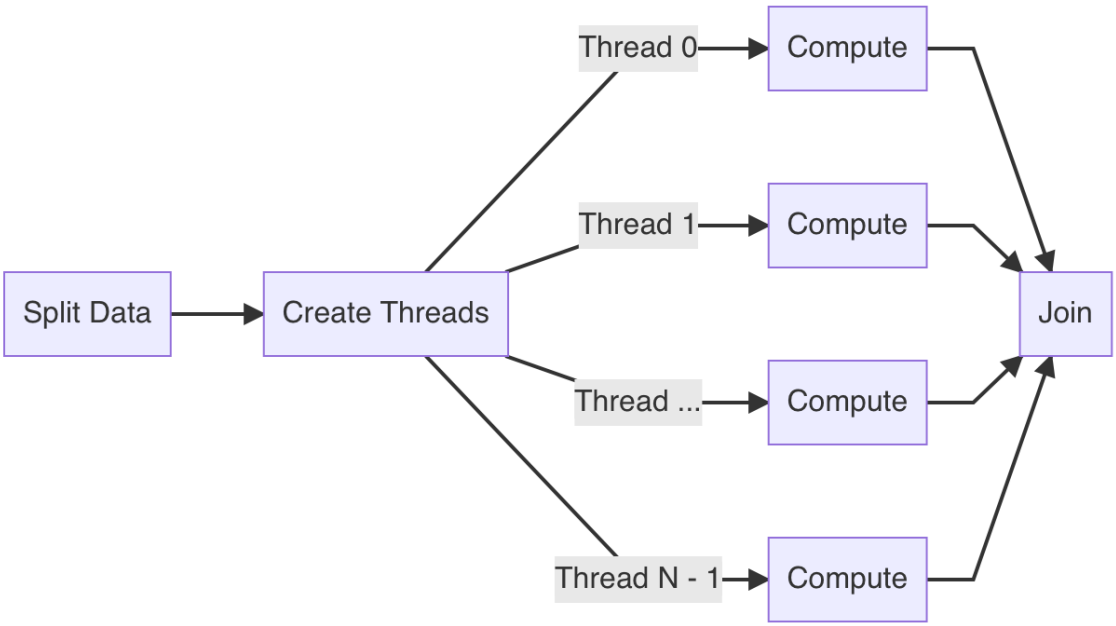
Parallel Mandelbrot Set Computation using Pthread

The following are the steps to perform parallel Mandelbrot set computation using Pthread:

1. Create a Point for each pixel in the image, and store all Points in the data array.
2. In total, m pixel Points are equally partitioned into n parts.
3. Create n child threads.
4. Inside each child thread, for each Point in its assigned data array, compute the point's color using the Mandelbrot set computation formula given in the introduction section.
5. Each child thread will join together.

The main idea is as follows. Firstly, total m pixel Points in the image are equally partitioned into n parts. Total n child threads are created and each child thread is responsible for 1 part of computation. In each child thread, the Mandelbrot set computation is performed for each Point in its assigned data array. Then each child thread will join together to output the final result.

This flow chart is illustrated in the following figure:



Implementation

Sequential Mandelbrot Set Computation

The implementation of Mandelbrot set computation directly follows the method mentioned above. The following is the core part of detailed implementation code:

```
void initData() {
    /*
    Intialize data storage.

    data =
    |   x1   |   x2   |  ...  |   xn   |
    |   y1   |   y2   |  ...  |   yn   |
    | color1 | color2 |  ...  | colorn |

    it represents a 2D array where each entry has a color attribute.

    x_i is in {0, 1, ..., X_RESN)}
    y_i is in {0, 1, ..., Y_RESN)}
    color_i is in {0, 1}

    */

    total_size = X_RESN * Y_RESN;
    data = new Point[total_size];
    int i, j;
    Point* p = data;
    for (i = 0; i < X_RESN; i++) {
        for (j = 0; j < Y_RESN; j++) {
            p->x = i;
            p->y = j;
            p ++;
        }
    }
}
```

```
void compute(Point* p) {
    /*
    Give a Point p, compute its color.
    Mandelbrot Set Computation.
    It is not necessary to modify this function, because it is a completed one.
    *** However, to further improve the performance,
    you may change this function to do batch computation.
    */

    Compl z, c;
    float lengthsq, temp;
    int k;

    /* scale [0, X_RESN] x [0, Y_RESN] to [-1, 1] x [-1, 1] */
    c.real = ((float) p->x - X_RESN / 2) / (X_RESN / 2);
    c.imag = ((float) p->y - Y_RESN / 2) / (Y_RESN / 2);

    /* the following block is about math. */

    z.real = z.imag = 0.0;
    k = 0;

    do {
        temp = z.real*z.real - z.imag*z.imag + c.real;
        z.imag = 2.0*z.real*z.imag + c.imag;
        z.real = temp;
        lengthsq = z.real*z.real+z.imag*z.imag;
        k++;
    } while (lengthsq < 4.0 && k < max_iteration);

    /* math block end */

    p->color = (float) k / max_iteration;

}
```

```
void sequentialCompute(){
    /* compute for all points one by one */
    Point* p = data;
    for (int index = 0; index < total_size; index++){
        compute(p);
        p++;
    }
}
```

Parallel Mandelbrot Set Computation using MPI

The implementation of parallel Mandelbrot set computation using MPI is divided into following steps:

1. Initialization and do distribution of input array elements.
2. Master process distributes data points to each slave process using MPI_Scatterv.
3. Perform local computation in each slave process.
4. Master process gather results from each slave process using MPI_Gatherv.

The following is the core part of detailed implementation code:

```
void worker(Point* my_data, int my_size) {
    Point* p = my_data;
    for (int index = 0; index < my_size; index++){
        compute(p);
        p++;
    }
}
```

```
// partition data to each process
int quotient = total_size / world_size;
int remainder = total_size % world_size;
int* send_counts = new int[world_size];
int* displacement = new int[world_size];

for (int i = 0; i < world_size; i++) {
    // number of elements allocated to each process
    send_counts[i] = quotient + (i < remainder ? 1 : 0);
}

displacement[0] = 0;
for (int i = 1; i < world_size; i++) {
    // displacement of each process
    displacement[i] = displacement[i - 1] + send_counts[i - 1];
}

int my_size = send_counts[rank];
Point* my_data = new Point[my_size]; // store elements of each process

// distribute elements to each process
MPI_Scatterv(data, send_counts, displacement, MPI_POINT, my_data, my_size, MPI_POINT, 0, MPI_COMM_WORLD);
worker(my_data, my_size);
// collect result from each process
MPI_Gatherv(my_data, my_size, MPI_POINT, data, send_counts, displacement, MPI_POINT, 0, MPI_COMM_WORLD);
```

Parallel Mandelbrot Set Computation using Pthread

The implementation of parallel Mandelbrot set computation using Pthread is divided into following steps:

1. Do partition of input array elements.
2. Create a thread pool containing n child threads using pthread_create.
3. Perform local computation in each child thread.
4. Each child thread joins together using pthread_join.

The following is the core part of detailed implementation code:

```c
int n_thd; // number of threads

typedef struct {
    //TODO: specify your arguments for threads
    int start; // array start position of whole data array
    int size; // size of array
    //TODO END
} Args;


void* worker(void* args) {
    //TODO: procedure in each threads

    Args* my_arg = (Args*) args;
    int start = my_arg->start;
    int size = my_arg->size;

    // for each point in assigned data array, compute its color
    Point* p = data + start;
    for (int index = 0; index < size; index++){
        compute(p);
        p++;
    }

    //TODO END
}
```

```c
initData();

//TODO: assign jobs

pthread_t thds[n_thd]; // thread pool
Args args[n_thd]; // arguments for all threads
int size = total_size / n_thd;
int remainder = total_size % n_thd;
// allocate data array for each thread
for (int thd = 0; thd < remainder; thd++){
    args[thd].start = thd * (size + 1);
    args[thd].size = size + 1;
}
for (int thd = remainder; thd < n_thd; thd++){
    args[thd].start = thd * size + remainder;
    args[thd].size = size;
}

for (int thd = 0; thd < n_thd; thd++) pthread_create(&thds[thd], NULL, worker, &args[thd]);
for (int thd = 0; thd < n_thd; thd++) pthread_join(thds[thd], NULL);
```

Execution

The following are the steps for execution:

1. Go to project2 folder, compile all files using “make all”.
2. Use “./seq [X resolution] [Y resolution] [max iteration]” to run sequential version without GUI.
3. Use “./seqg [X resolution] [Y resolution] [max iteration]” to run sequential version with GUI.
4. Use “mpirun -np [number of processes] ./mpi [X resolution] [Y resolution] [max iteration]” to run parallel version using MPI without GUI.
5. Use “mpirun -np [number of processes] ./mpig [X resolution] [Y resolution] [max iteration]” to run parallel version using MPI with GUI.
6. Use “./pthread [X resolution] [Y resolution] [max iteration] [number of threads]” to run parallel version using Pthread without GUI.

7. Use "./pthread [X resolution] [Y resolution] [max iteration] [number of threads]" to run parallel version using Pthread with GUI.

The following are the steps to do experiments in hpc:

1. Go to project2 folder, compile all files using "make all".
2. Config the task template batch file by specifying job name, nodes number, process number, number of CPU cores per process, total memory limit, time limit, and partition name shown as the following example.

   #!/bin/bash

   #SBATCH –job-name=119020059_seq_{size} # Job name

   #SBATCH –nodes=1 # Run all processes on a single node

   #SBATCH –ntasks=1 # number of processes = 4

   #SBATCH –cpus-per-task=1 # Number of CPU cores per process

   #SBATCH –mem=1000mb # Total memory limit

   #SBATCH –time=00:60:00 # Time limit hrs:min:sec

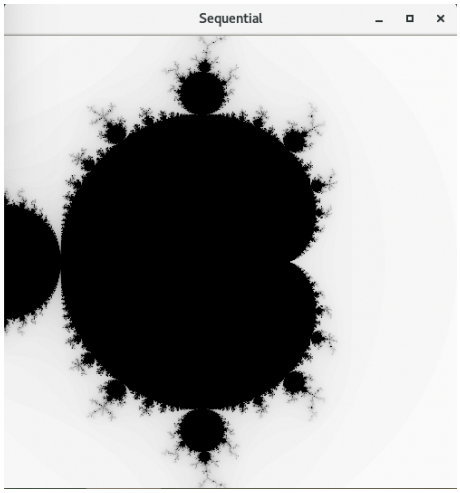
   #SBATCH –partition=Debug # Partition name: Project or Debug (Debug is default)

3. SSH to hpc and run "sbatch [batch file path]" to submit the job.
4. Collect running output file "slurm-[num].out".

Sample Outputs

Sequential Mandelbrot Set Computation

X resolution = 1000, Y resolution = 1000, Max iteration = 100, Number of processes = 1. The following is the output:




```
[csc4005@localhost project2]$ ./seq 1000 1000 100
Student ID: 119020059
Name: Xinyu Xie
Assignment 2 Sequential
Run Time: 0.209522 seconds
Problem Size: 1000 * 1000, 100
Process Number: 1
```

Parallel Mandelbrot Set Computation using MPI

X resolution = 1000, Y resolution = 1000, Max iteration = 100, Number of processes = 4. The following is the output:

```
[csc4005@localhost project2]$ mpirun -np 4 ./mpi 1000 1000 100
Student ID: 119020059
Name: Xinyu Xie
Assignment 2 MPI
Run Time: 0.110513 seconds
Problem Size: 1000 * 1000, 100
Process Number: 4
```

Parallel Mandelbrot Set Computation using Pthread

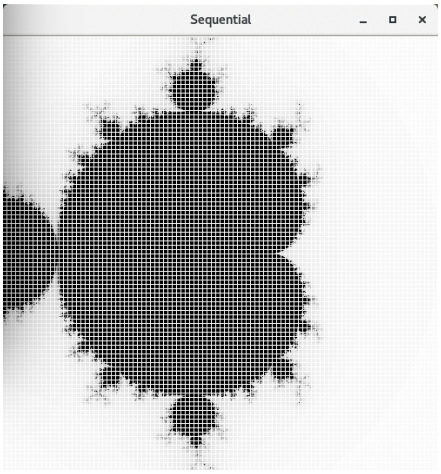
X resolution = 1000, Y resolution = 1000, Max iteration = 100, Number of threads = 4. The following is the output:

```
[csc4005@localhost project2]$ ./pthread 1000 1000 100 4
Student ID: 119020059
Name: Xinyu Xie
Assignment 2 Pthread
Run Time: 0.112739 seconds
Problem Size: 1000 * 1000, 100
Thread Number: 4
```

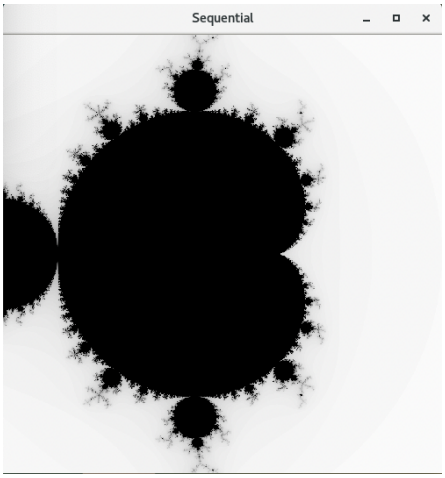
Analysis and Results

Size of output images

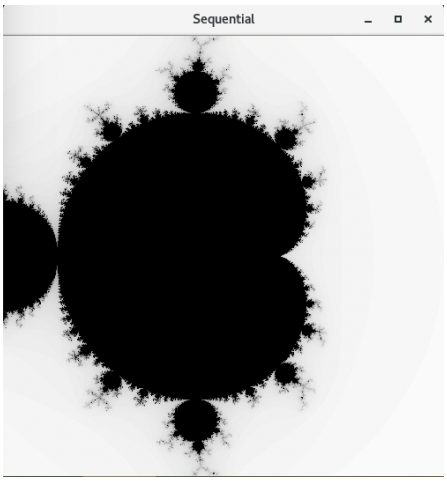
X resolution = 400, Y resolution = 400, Max iteration = 100. The following is the output:



X resolution = 800, Y resolution = 800, Max iteration = 100. The following is the output:



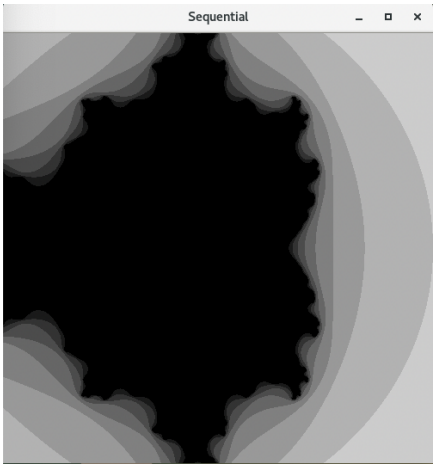
X resolution = 1600, Y resolution = 1600, Max iteration = 100. The following is the output:



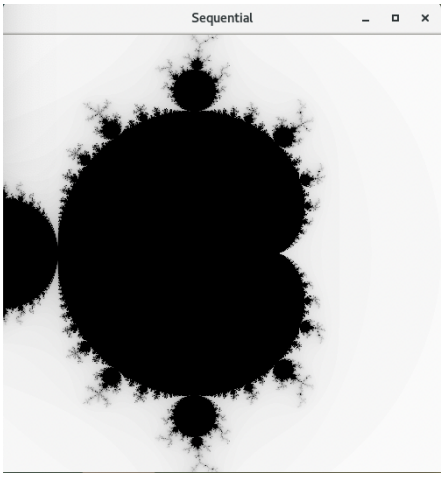
Since the size of output images are defined by the input x-resolution and y-resolution, adjusting the size will output images with different fineness. When the size of output image is small, the image is less fine. And when the size of output image is large, the image is finer.

Maximum iteration numbers

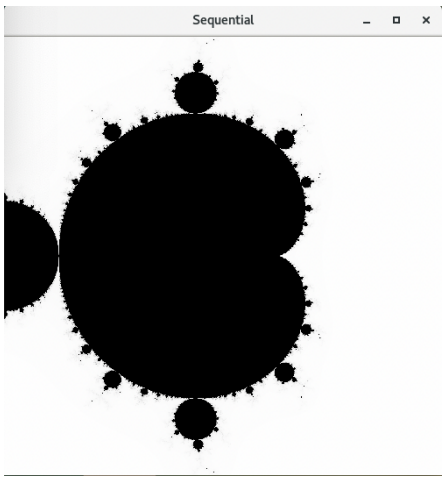
X resolution = 1000, Y resolution = 1000, Max iteration = 10. The following is the output:



X resolution = 1000, Y resolution = 1000, Max iteration = 100. The following is the output:



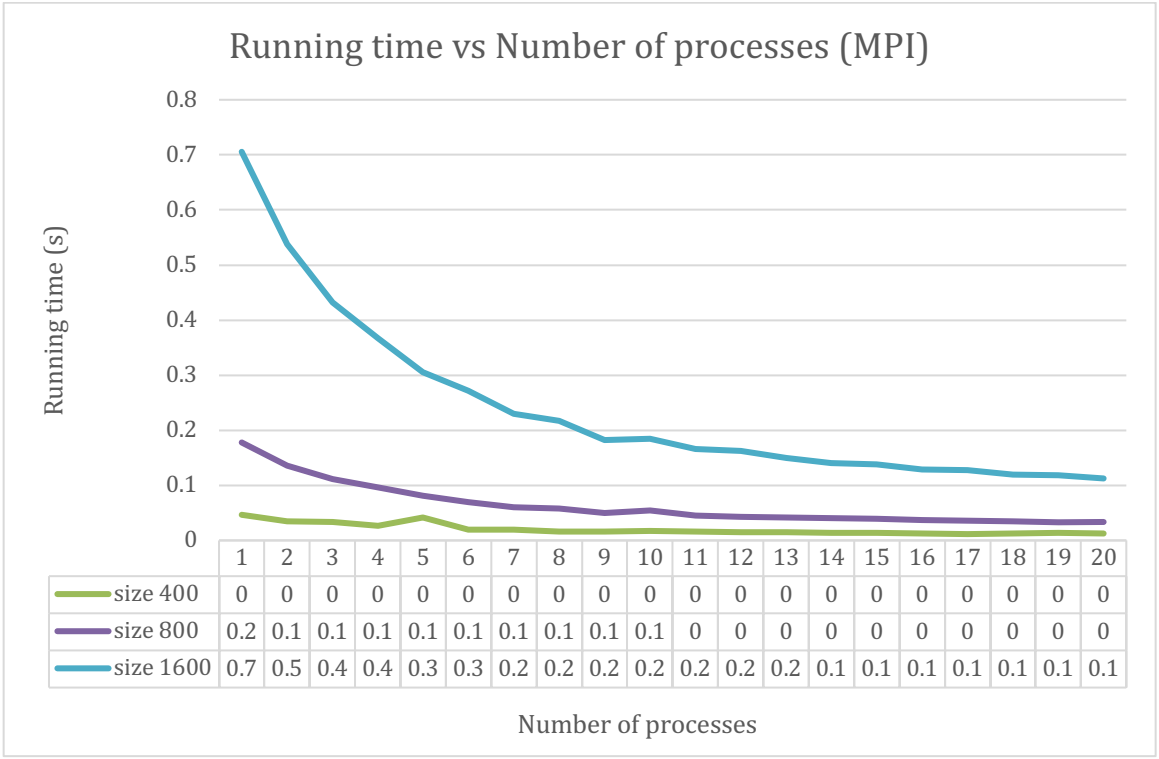
X resolution = 1000, Y resolution = 1000, Max iteration = 1000. The following is the output:



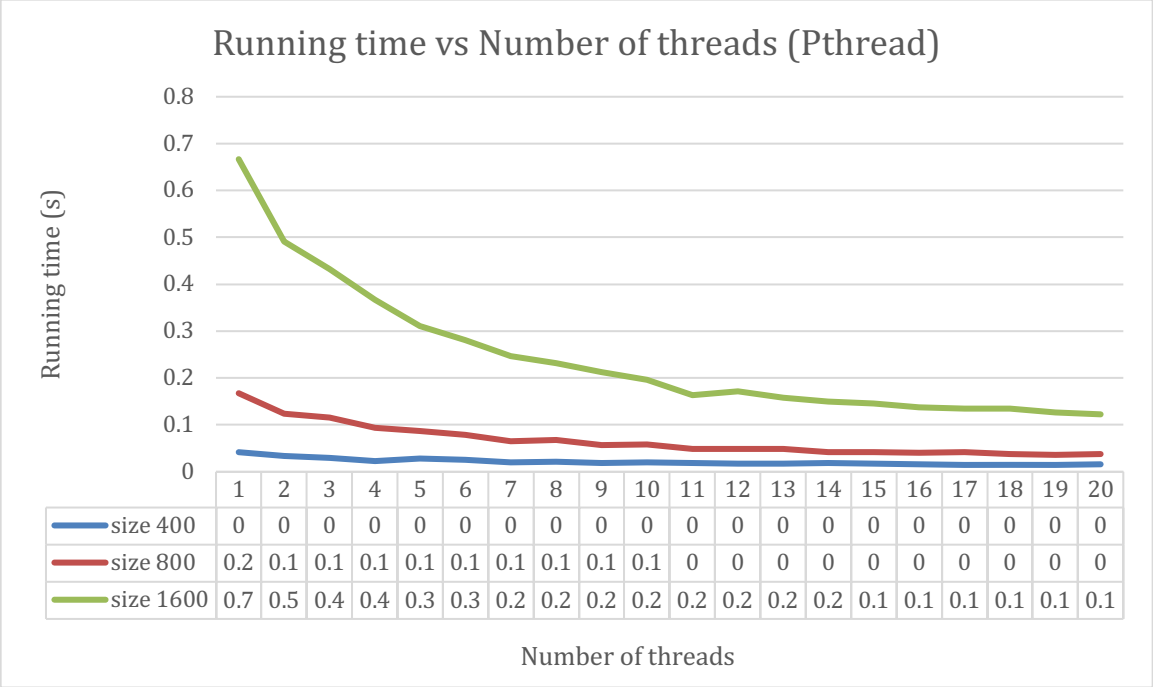
Maximum iteration number is set inside the Mandelbrot set computation. Different maximum iteration numbers will result in slightly different output images. When the maximum iteration number is small, there are more points shown in the output image. And when the maximum iteration number is large, there are less points shown in the output image while the difference is small.

Running time vs. Number of processes/threads

We compare parallel version of MPI and Pthreads with number of processes/threads = 1, 2, 3, … 20 and image size = 400 * 400, 800 * 800, 1600 * 1600. The comparison is shown as follows:



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
size 400	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
size 800	0.2	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0	0	0	0	0	0	0	0	0	0
size 1600	0.7	0.5	0.4	0.4	0.3	0.3	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.1	0.1	0.1	0.1	0.1	0.1	0.1



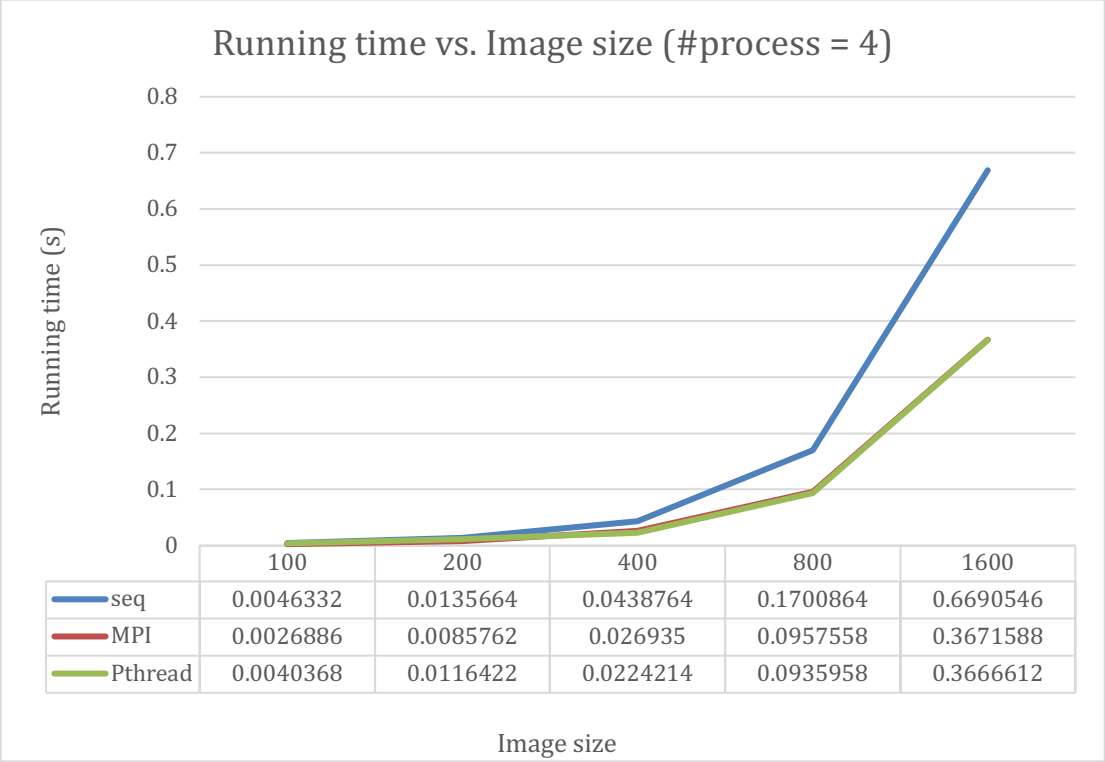
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
size 400	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
size 800	0.2	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0	0	0	0	0	0	0	0	0	0
size 1600	0.7	0.5	0.4	0.4	0.3	0.3	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.1	0.1	0.1	0.1	0.1	0.1

The above images show both the MPI version and Pthread version of the relationship between running time and number of processes/threads with different image size. From the figure above, I found that when the number of processes/threads increases, the running time decreases with slight fluctuations. Since when the number of processes is larger each process is assigned with less amount of work, thus the running time taken for a single process becomes less. This illustrates that using the parallel computing techinique, the computation time could be reasonably decreased.

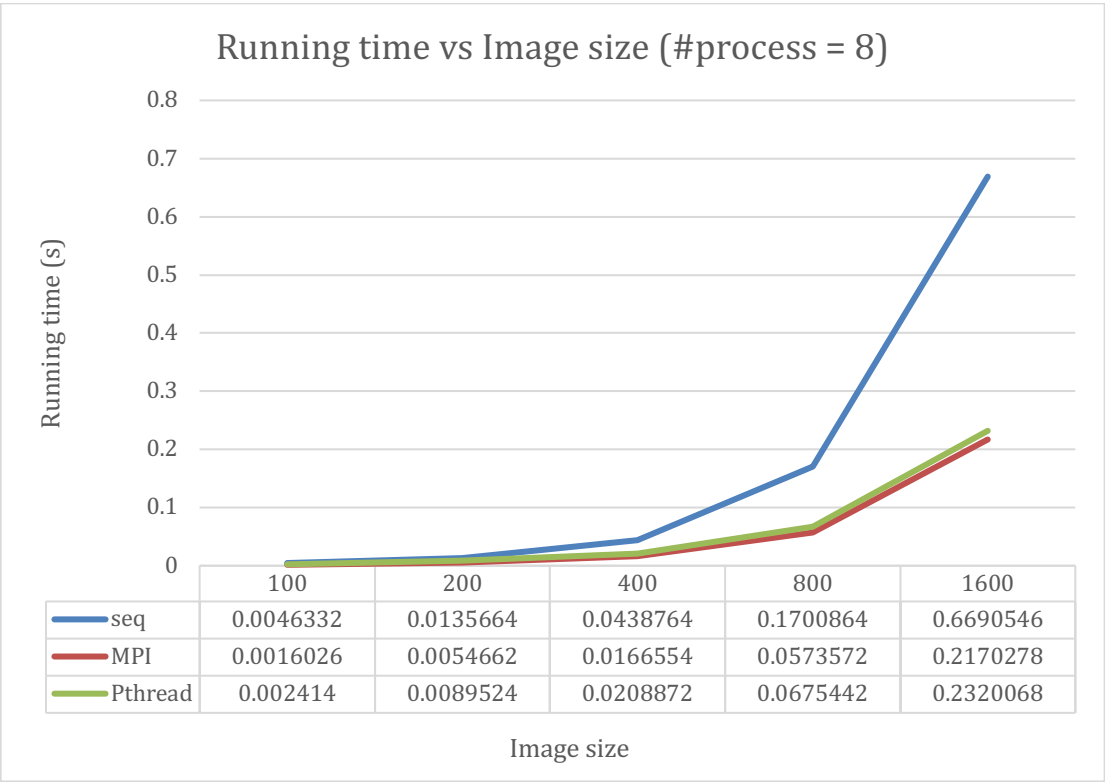
When the number of processes/threads is very large, the running time is roughly stable with little or even no decrease in running time as number of processes/threads increases. Since when number of processes/threads is large enough, as it increases, the decrease in computation time is very small while having extra cost for process/thread creation, data scatter and gather or thread synchronization. Thus, the running time is roughly stable.

Sequential vs. MPI vs. Pthread

We compare sequential version, parallel version of MPI and Pthreads with image size = 100 * 100, 200 * 200, 400 * 400, 800 * 800, 1600 * 1600, by fixing number of processes/threads = 4 or 8. The comparison is shown as follows:



	100	200	400	800	1600
seq	0.0046332	0.0135664	0.0438764	0.1700864	0.6690546
MPI	0.0026886	0.0085762	0.026935	0.0957558	0.3671588
Pthread	0.0040368	0.0116422	0.0224214	0.0935958	0.3666612



	100	200	400	800	1600
seq	0.0046332	0.0135664	0.0438764	0.1700864	0.6690546
MPI	0.0016026	0.0054662	0.0166554	0.0573572	0.2170278
Pthread	0.002414	0.0089524	0.0208872	0.0675442	0.2320068

The above images show the sequential version, MPI version and Pthread version of the relationship between running time and output image size with fixed number of processes/threads. From the figure above, I found that keeping number of

processes/threads constant, when the size of output image is small, the running time is less. Since when the output image size is large, total number of data points needed to be computed is larger, thus longer computation time is needed. The running time is roughly quadratic with image size from the graph, since when the size doubles, number of data points is multiplied by 4 considering both x-resolution and y-resolution double. Thus, the computation time should be roughly 4 times.

Moreover, I found that MPI and Pthread versions have much less running time compared with sequential version. This illustrates that parallel computing could indeed reduce much computation time. The running time of MPI version and Pthread version is roughly the same when number of processes equals to number of threads. The MPI version runs slightly faster than Pthread version with same number of processes/threads. This is partly because that MPI initializes all the processes at the beginning and this time is not counted in the computation. However, in Pthread version, all the children threads are created inside the computation part, which takes some time. This may also due to the dispatch of Pthread so that one thread runs slower than other threads. And the thread synchronization costs extra time for Pthread version.

Conclusion

In this project, the sequential version, parallel version using MPI, and parallel version using Pthread of Mandelbrot set computation are implemented and their proformances are also analyzed. We can conclude from this project that when the number of processes/threads increases, the running time decreases with slight fluctuations. When the number of processes/threads is very large, the running time is roughly stable with little or even no decrease in running time as number of processes/threads increases. Moreover, keeping number of processes/threads constant, when the size of output image is small, the running time is less. The running time is roughly a quadratic function of image size.

Through this project, I learned much knowledge regarding the parallel programming and also how to use the mpi and pthread library. I also enjoy a lot when conducting experiments for different image resolutions, different versions of implementation, and exploring the relationship between the running time and the number of processes/threads.