

香港中文大學(深圳)

The Chinese University of Hong Kong, Shenzhen

CSC4005 Distributed and Parallel Computing

Report 2:
**A Comprehensive Implementation and Analysis of
Mandelbrot Set Computation by MPI and thread**

Submitted by:

Changwen LI

118010134

Submission date:

30th of October 2021



香港中文大學(深圳)

The Chinese University of Hong Kong, Shenzhen

Table of Contents

1. Introduction	3
2. Methods.....	3
3. Analysis.....	7
4. Discussion.....	10
5. Conclusion.....	11

A Comprehensive Implementation and Analysis of Mandelbrot Set Computation by MPI and thread

1. Introduction:

This assignment requires students to implement Mandelbrot Set by MPI (Message Passing Interface) and thread. The report consists of 4 parts. In the first part, main concepts are introduced. The second part introduces the program architecture and the procedures to execute the program. In the third part, the analysis part, we test the programs with different number of processes, threads and different length of data. We also compare the performance of implementation of MPI, thread and sequential method. In the fourth part, which is the discussion part, we compare the performance of implementation of static threading and dynamic threading.

1.1. Introduction to Mandelbrot Set

Set of points in a complex plane that are quasi-stable (will increase and decrease, but not exceed some limits) when computed by iterating the function: $z_{k+1} = z_k^2 + c$. when z_{k+1} is the kth iteration of the complex number and is a complex number giving the position of the point in the complex plane. The initial value for is zero. The iterations continued until the magnitude of is greater than a threshold or the maximum number of iterations have been achieved. Then we take the real part and imaginary part of the number as the x-coordinate and y-coordinate of the graph. We may view the iteration as several independent processes to calculate the value of the row number of the graph (i.e. the imaginary part of the calculation) so that we can exploit the parallel nature of Mandelbrot Set calculation.

1.2. Introduction to MPI

MPI (Message Passing Interface) is a protocol for passing messages between processes which is widely used in information transferring in parallel and distributed computing. In this project, the block point-to-point communication is used. The blocking sending, receiving, broadcasting and other information passing schemes provided by MPI are used.

1.3. Introduction to thread

Thread is the basic smallest sequence of programmed instructions that can be managed by a scheduler, which is typically a part of the operating system. Threads are executed concurrently with sharing resources such as memory and global variables. Compared to process, thread saves more resources in general. To program with threads, library `<thread>`, the standard library of C++ 12 is used in this project. There are 2 versions of thread codes in this project. One of them is the static version, where all threads are scheduled by program to do the jobs assigned by the programs before compiling and executing. Another version is the dynamic one, where threads' workload is not assigned by programs beforehand. They are assigned with jobs automatically once they are free. More details are presented in the discussion part.

2. Methods

2.1. Program implementation

The program is implemented in 3 ways (MPI, static threading, dynamic threading). If the sequential paradigm is taken into consideration, then this project provides 4 implementations of Mandelbrot Set calculation.

For the MPI implementation, the configuration such as size is sent to other processes by process 0. The configuration data is written in an array so that they can be sent by MPI. The code of this part is presented in figure 2.1.1. After all the other processes receive the configuration, the calculation starts. When the data has finished calculation, they are sent to process 0. Then the process 0 will start to write all the data in the canvas as the input. A tricky thing to pay attention to is, the MPI_Send may not necessarily provide ideal block transmission if the data size is large. Sometimes the program will continue once the MPI_Send function sends data to the buffer rather than other processes. Thus, here we add another variable as the flag to synchronize the data transmission. When process 0 finishes loading received data on the variable canvas, we send a signal to other processes to continue next iteration of graph computing and drawing. The structural graph is attached in figure 2.1.2.

```
// send msg about variables to other proc, maintain the same value of variables for all procs
int *msg_send = new int[5];
msg_send[0] = size;
msg_send[1] = scale;
msg_send[2] = center_x;
msg_send[3] = center_y;
msg_send[4] = k_value;

for (int k=1; k<rank_num; k++){
    MPI_Send(msg_send, 5, MPI_INT, k, 888888, MPI_COMM_WORLD);
}
```

Figure 2.1.1: Screenshot of configuration sending in MPI

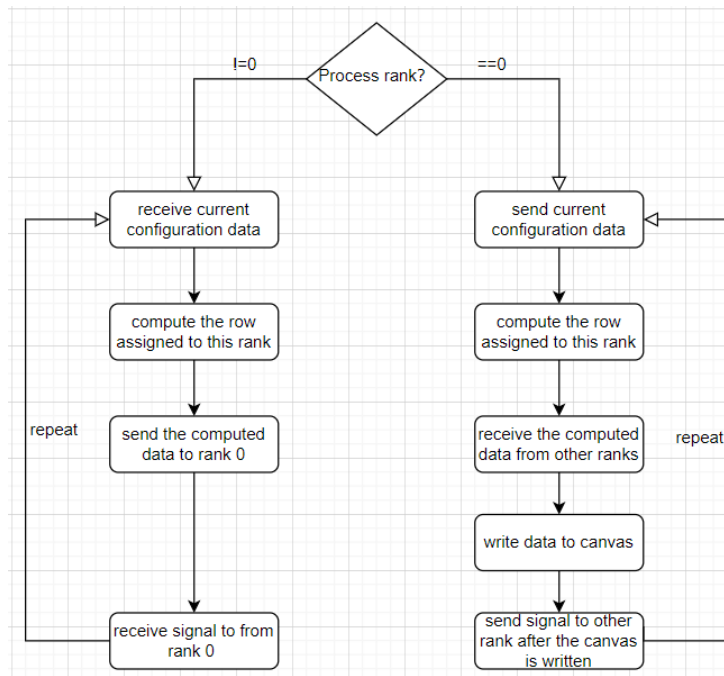


Figure 2.1.2: Brief architecture of MPI implementation

For the static threading programming, we assign job to each threads in the codes. Each thread is assigned to $\left\lfloor \frac{m}{n} \right\rfloor$ rows to calculate. For robust design, the process with rank 0 will calculates the

remainder part of rows as well. Here, we use mutex to denote the current available rows. We use a lock to ensure the atomicity of the current available rows. Thus, only one thread can modify this number at the same time. The codes of this part is attached in figure 2.1.3.

```
// get the current available row idx for thread to calculate.
// use mutex to issue the atomic operation.
int local_row_idx=0;
lock_mutex.lock();
global_row_idx++;
if (global_row_idx>=size){
    global_row_idx=0;
}
local_row_idx = global_row_idx;
lock_mutex.unlock();
```

Figure 2.1.3: Screenshot of mutex in threading

In the dynamic threading programming, we do not assign jobs to threads in advance. On the contrary, we work to the free thread. The advantage of dynamic paradigm will be discussed later in discussion part. Here we mainly focus on the implementation. First of all, we use the same techniques of locking the current uncalculated row number mentioned before. Second of all, we use another array with the size of threads to record which thread has finished its job. The value of all the members in this array is 0, denoting they are all available. After the main program finds the available one, it will change the number to 1 before running that thread. After that thread has finished its task, it will modify the value in the array to 0 again to show that this process is available again. When the main program finds a thread already finished its job, it will join it to free the thread. The codes of this part is shown in figure 2.1.4. The brief architecture of dynamic scheduling by thread is shown in figure 2.1.5.

```
int already_iter = 0;
while (already_iter<size){
    for (int i=0; i<num_threads; i++){
        if (occupied_threads[i]==0){
            already_iter++;
            if (new_threads[i]==1){
                thread_arr[i].join();
            }
            occupied_threads[i]=1;
            new_threads[i] = 1;
            thread_arr[i]=std::thread(calculate_thread, occupied_thre
            break;
        }
    }
}

// before exiting the program, ensures that all threads has joined.
for (int i=0; i<num_threads; i++){
    if (thread_arr[i].joinable()){
        thread_arr[i].join();
    }
}
```

Figure 2.1.4: implementation of dynamic method in threads

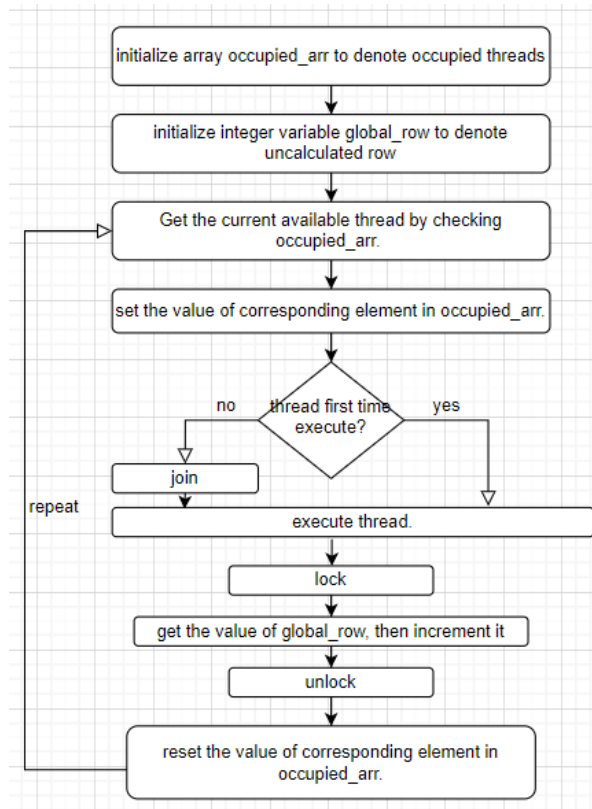


Figure 2.1.2: Brief architecture of MPI implementation

2.2.Procedures to execute the program

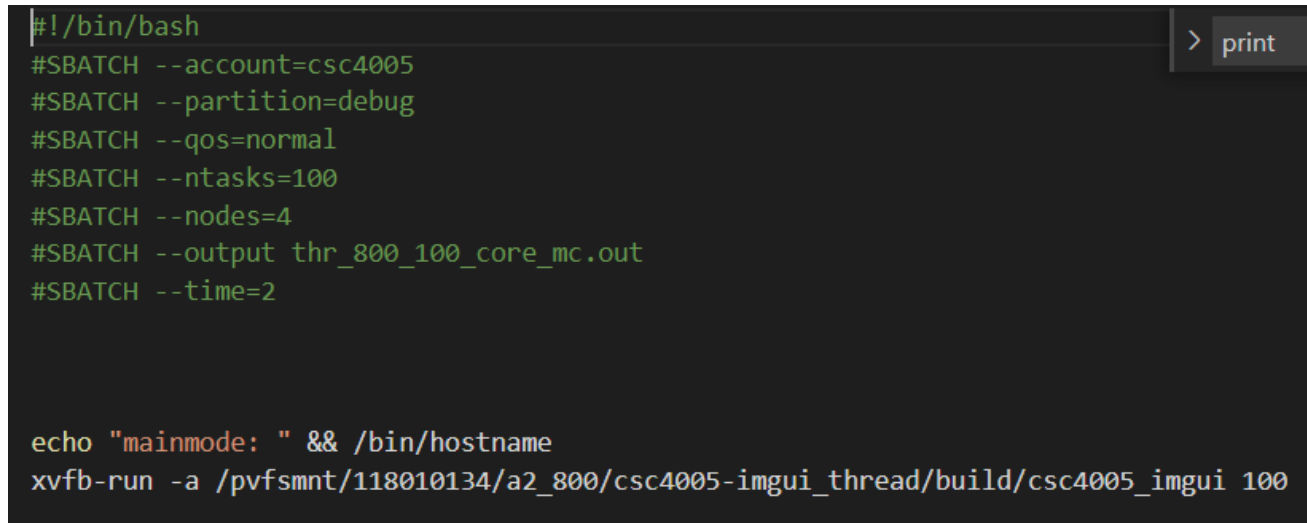
The project is based on templates provided by TAs and it is already compiled. There are 4 documents, which provides MPI, static threading, dynamic threading and sequential.

The source files are under the `/src` directory. The executables are under each `/build` directory. To execute the programs, you may use either local host or the Slurm platform.

If you are using a localhost, the command to run the executables for MPI version is `$mpirun -np num_of_process ./csc4005_imgui$`. The command to run the executables for thread version is `$/csc4005_imgui thread_num$`. Please notice that the command is slightly different from the original version provided on BB. I modify the program so that the program can take an integer as input in command. If you forget to type an integer input, the program can still be executed with a default number of threads. The command to run the executables for sequential version is `$/csc4005_imgui$`. You are recommended to execute the program on local host if you want to see the graphic results because the graphic user interface is extremely slow on Slurm.

We recommend you to execute the programs on Slurm if you only want to obtain the experiment data. If you are using Slurm, then command approach is not the best approach to execute them on Slurm. The recommended one is using batch approach. There is a batch file (`run_a2.sh`) in each folder. You can directly execute them if you want. Please modify the parameters in the batch file to suit your requirements. The number after `ntasks` indicates the number of processors, the number after `nodes` indicates the number of nodes you want. You can also modify the information after `output` to denote the name of the file storing the printed information. The command starts with `"xvfb"` is the same as the commands in localhost (except there is no need to state the number of processes).

Modify it if it is necessary for you. An example of batch file is shown in figure 2.2.1. After modifying the batch file, use command `$sbatch batch_name.sh$` to run the batch files (You may need to wait in queue).



```
#!/bin/bash
#SBATCH --account=csc4005
#SBATCH --partition=debug
#SBATCH --qos=normal
#SBATCH --ntasks=100
#SBATCH --nodes=4
#SBATCH --output thr_800_100_core_mc.out
#SBATCH --time=2

echo "mainmode: " && /bin/hostname
xvfb-run -a /pvfsmnt/118010134/a2_800/csc4005-imgui_thread/build/csc4005_imgui 100
```

Figure 2.2.1: An example of batch file for executable

2.3. Methods of experiments and data collection

The project tests the performance of the calculations with different number of processes/threads and different data size. Please notice that all data about threading refers to the static threading (because my dynamic thread part was finished only 20 hours before deadline). There are mainly three set of experiments conducted in this project.

In the first set of experiment, the performance of different data size with the same number of processes/threads are tested. The fixed number of processes/threads is 5, with data size of 400 pixels, 800 pixels and 1600 pixels. In the second set of experiment, different number of threads/processes are used to test on same size of data. 2 threads/processes, 4 threads/processes, 8 threads/processes, 16 threads/processes and 32 threads/processes are used to test on 800 pixels. In the third set of experiment, parallel computing with MPI and thread and sequential computing are being compared.

3. Analysis

3.1. Speed-up analysis:

Assume that the workload is fixed. We can make this assumption since most of the experiments are conducted with same data size. Then we may use Amdahl's Law to approximate the speed up.

According to Amdahl's Law, we have $S(n) = \frac{n}{1+(n-1)f}$ where n represents the number of nodes and f represents the percentage of sequential part. When the parallel part of the codes takes the majority, the speed up should be closed to n .

3.2. Experiment result analysis

3.2.1. Performance of different data size with the same number of processes

In this experiment set, the number of processes is assigned to be a fixed number. The number of assigned processes/threads is 5. Then, test data with 400 pixels, 800 pixels and 1600 pixels are tested.

The raw data of threads is shown in figure 3.2.1.1. The raw data of processes is shown in figure 3.2.1.2. The speed of execution drops when the data set gets larger. It is clear that the calculation speed drops as the data size increases. The reason is that each pixel requires more calculation when the size increases. For instance, when the size is 800, each pixels require 800 calculations. When the size is 1600, 1600 calculations are needed.

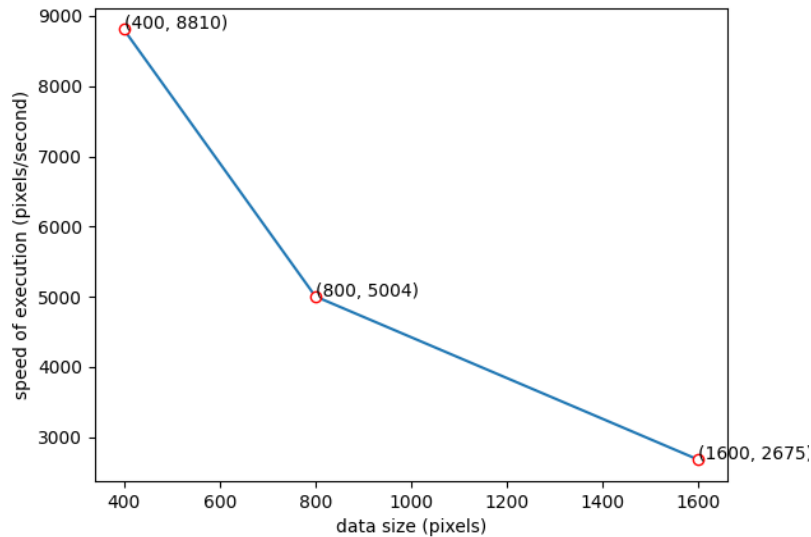


Figure 3.2.1.1: Execution time of different data size with 5 threads

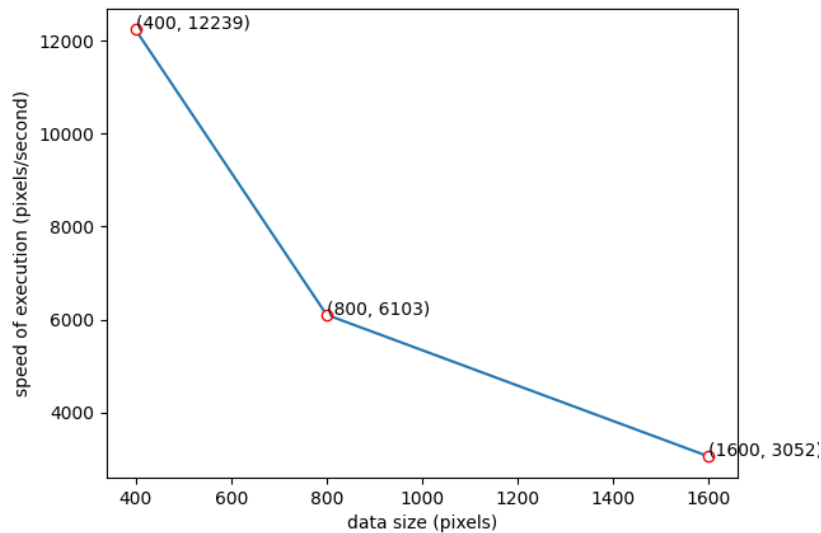


Figure 3.2.1.2: Execution time of different data size with 5 processes

3.2.2. Performance of different number of processes/threads with the same data size

In this experiment set, the data size is assigned to be a fixed number. The data size is 800 pixels. Then, different number of cores are used to test on same size of data. 2 threads/processes, 4 threads/processes, 8 threads/processes, 16 threads/processes and 32 threads/processes are used to test on 800 pixels. The raw data is shown in figure 3.2.2.1 for process and the raw data for process is

shown in figure 3.2.2.2. The experiment results fit the theoretical analysis of Amdahl's Law. The execution speed increases as the number of parallel units increase.

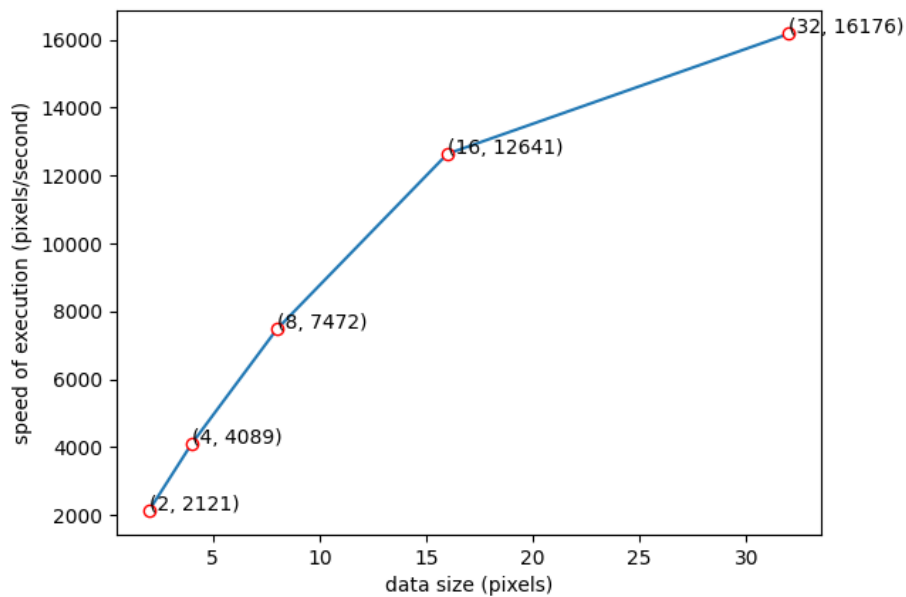


Figure 3.2.2.1: Execution time of different number of threads size with 800 pixels

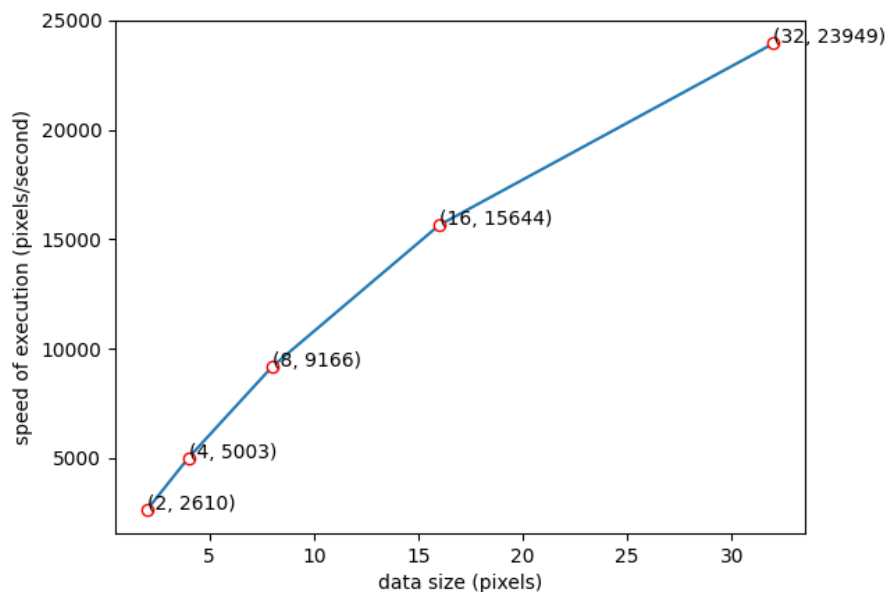


Figure 3.2.2.2: Execution time of different number of processes size with 800 pixels

3.2.3. Performance comparison between sequential programming and parallel programming with MPI and thread

We set the data size as 800. The sequential version of programs is provided in BB. The execution speed is 1417 pixels per second. For the parallel version with 5 threads and 5 processes, the speed is 5004 pixels per second and 6103 pixels per second. The speed-up for parallel version is around 4.3

when the parallel unit is 5. The speed up is not 5 because some sequential codes are included in program such as the variables initialization and graph initialization.

The MPI version and thread version shares similar speed and MPI version is faster. The reason why MPI version is faster when the thread numbers is equivalent to the process number results from the architecture and mechanism of thread scheduling and process scheduling. For process scheduling, each core is responsible for only one process. However, it is possible that one core is responsible for several threads because the thread is scheduled by a scheduler provided by the OS. In other words, we can ensure that all processes run simultaneously. However, some threads run concurrently. Therefore, the process method exploits more parallel nature than the thread method.

4. Discussion

In discussion, we analyze the static scheduling and dynamic scheduling.

In static scheduling, we divide the whole job into several sub-tasks and assign them to different process or threads. After each of them finished their own task, we integrate their results and get the final output. Each process or thread is responsible for same amount of work but different amount of workload. For instance, if the size is 4 and the thread/process number is 2, then each process will calculate 2 rows of data, which indicates the amount of tasks is same for each process/thread. However, the work may have different workload since some work requires more computation.

In dynamic scheduling, the sub-tasks are assigned to threads/processes dynamically once they are available. In other words, in dynamic scheduling, some threads/processes may calculate more rows than the other. However, the workload for each thread/process will be much more balanced. The dynamic scheduling saves more computation time because threads/processes can be used more efficiently. However, dynamic scheduling may have a longer overhead time and it is more difficult to implement.

An experiment is made to compare the performance of static scheduling and dynamic scheduling. The result is shown in figure 4.1. We can see that the performance of static scheduling is better when the data size is relatively small. The reason is that the overhead time in dynamic scheduling is higher since it needs to check which thread is empty. When the data size increases, the imbalance of calculations takes up the most of the execution time, then the dynamic scheduling is faster.

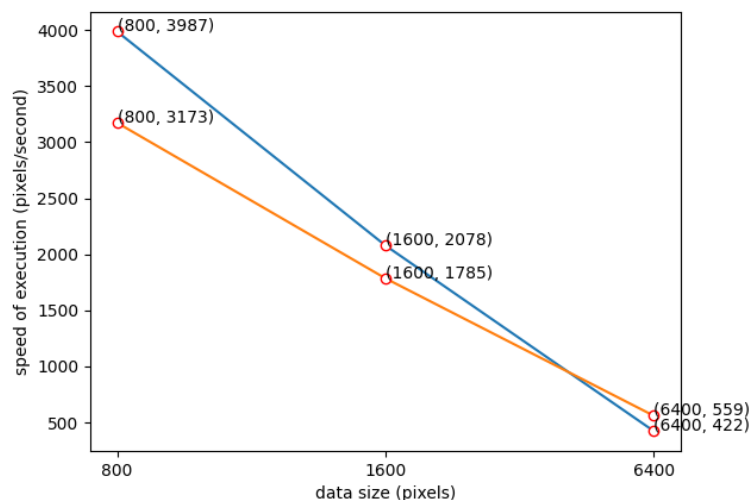


Figure 4.1: Comparison between dynamic scheduling and static scheduling

5. Conclusion

The project implements Mandelbrot Set Computation. The report has analyzed the performance of it with different configurations. Generally speaking, the experimental results fit the theoretical analysis. The comparison between dynamic scheduling and static scheduling can be seen in discussion.