Assignment 2
CSC4005 Distributed and Parallel Computing

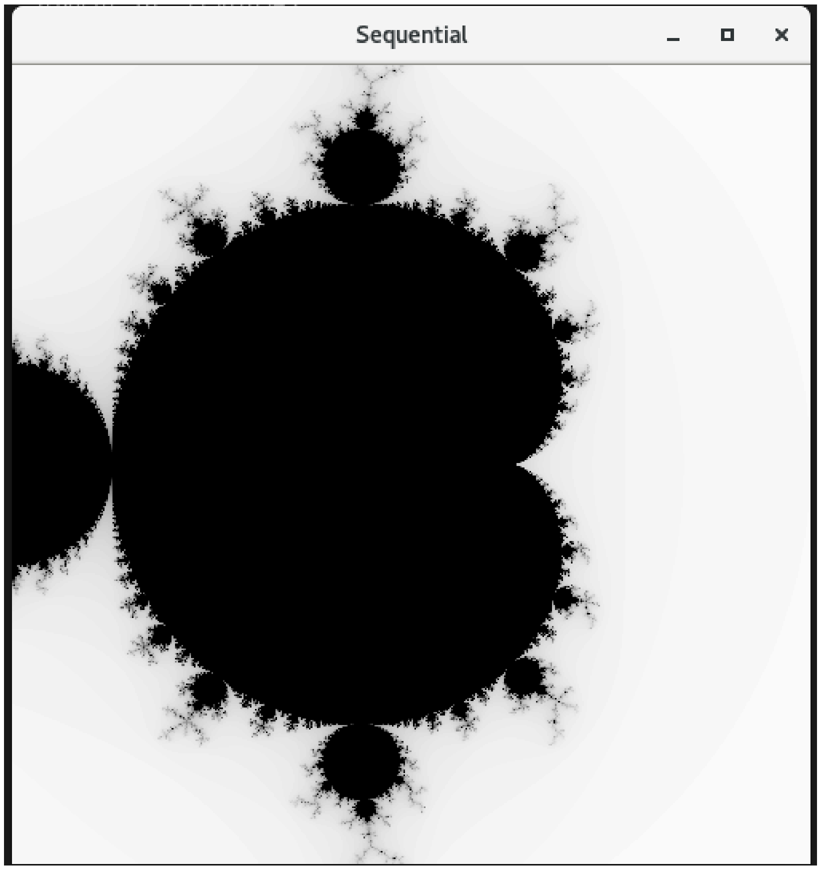
Bernaldy Jullian – 119010520
School of Data Science
The Chinese University of Hong Kong, Shenzhen

I. Introduction

Until now, human still struggles with so many problems from the area of pure mathematics to a more practical problems such as economy practices. Fortunately, in this era of technology, the emergence of computer can solve many problems, especially those problems that may need to be solved brute-forcedly. However, despite the rapid development of computer performance every year, some problems may still need to be solved with a better performance. Thus, the parallel computing is introduced to overcome this problem. In the development of parallel computing, human created many approaches that can possibly offer a better method of parallelization. However, the result is another new way of parallelization that has its own advantages and disadvantages. This paper will report the result of experiments between two parallel computing methods, namely PThread and MPI.

II. Method

In this experiment, we will use Mandelbrot Set as the problem that we want to compute. In order to reduce any unwanted factors, we will measure the performance of each implementation based on only its compute time, excluding the initialization of the program (such as code parts that provide graphical support for ensuring the correctness of the code). The output of all implementations must be an array containing Boolean values that represent black (zero) and white (one) of the picture.



Gui program that generates the Mandelbrot set fractal

Sequential

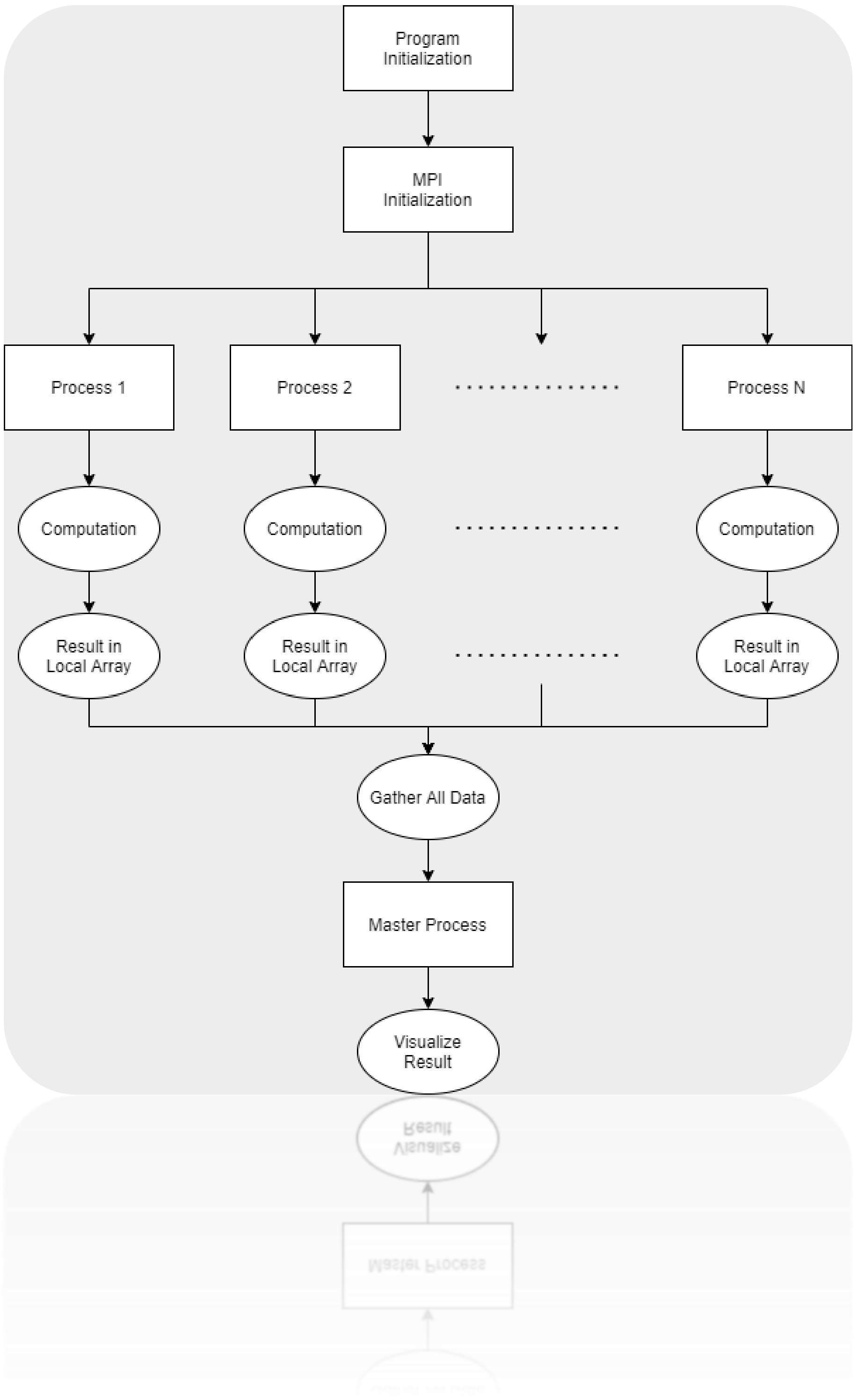
The sequential program is given along with the assignment template and format. The program initializes the data according to the desired resolution and max iteration value. Then, it uses a for loop that will compute the recurrence of every pixel and color it black or white accordingly. The pixels and data generation are all stored in an array of a custom data structure that holds the coordinates of the pixel in the complex plane alongside its color.

MPI

This parallelization technique exploits some processors in a cluster to work together in solving a problem. Since processors are usually separated, they are most likely to be designed with a distributed-memory structure. So, there will be no common memory among the processors. Thus, this technique needs programmers to explicitly invoke the communications between processors in order to have data-sharing among them.

In general, the implementation of MPI is initiated with the distribution of the problem from the main processor (called master) to any other processors. However, in this experiment, our problem input is generated directly from a loop that represent the two-dimensional image that we want to achieve. Thus, after some program initializations (such as for graphical purposes), the program immediately runs the loop (which starts the time measurement). One thing to notice that, although MPI technique is based on a distributed-memory structure, the global variables of the program (especially the constant one) is still accessible by all processes. The reason is when the MPI is initiated, it copies all the available global variables (and possibly function too) to each process. Hence, each process may have the same content for each global variable, but an update of global variables in one process will not change the global variable of the other processes (we can imagine this like all processes are independent programs with same global variables initialization).

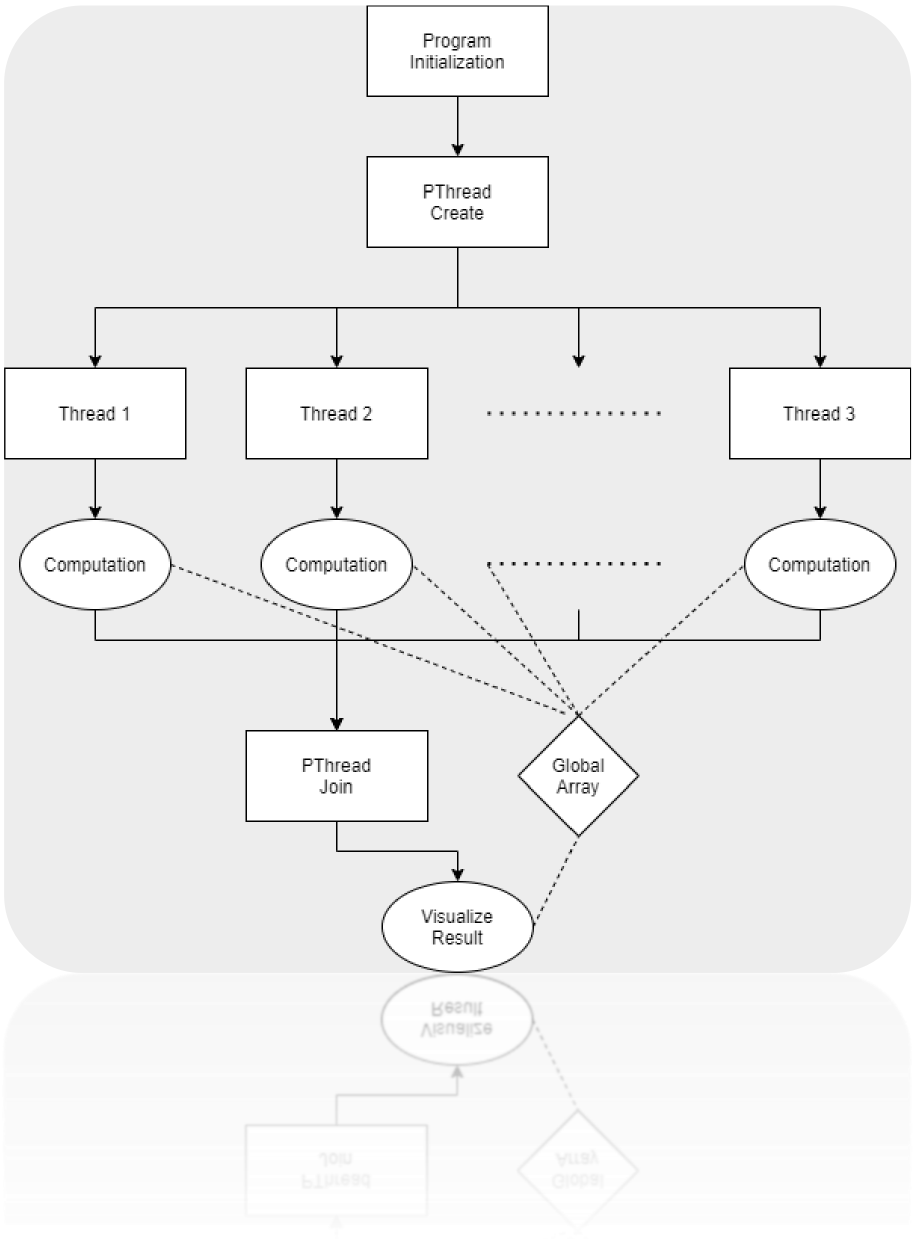
Thus, we can use the feature above so that each process knows what portion of the program it will compute. This is done by creating global variables for the size of the image and some configurations for the Mandelbrot Set computation (such as maximum iteration and calculation threshold). Then, in the loop, each process will run a similar computation that is done by the sequential program but with a smaller size (it must be smaller if we assume that the sequential and MPI implementations are doing the same-sized image with the same configurations). The result of each process will also be stored into a pre-declared local array. After finishing all of the computations, each process is faced with a barrier that delays every process until all processes are ready. After passing the barrier, the gather communication among the processes are explicitly triggered. This communication instructs all processes to combine their local arrays into one big array in the master process. This communication ends the time measurement since we already have a complete Mandelbrot Set solution in the big array located in the master process. In the end, we can request the master process to visualize the array it has.


Program Initialization
MPI Initialization
Process 1
Process 2
Process N
Computation
Computation
Computation
Result in Local Array
Result in Local Array
Result in Local Array
Gather All Data
Master Process
Visualize Result

PThread

This method of parallelization is utilizing the threads that exist in the computer processor. Unlike MPI, this method does not need to be run on a cluster and it can directly be run on a personal computer. Besides, the threads that are employed by this method are based on shared-memory structure. That means that all threads can have common variables, and so, they can directly do memory-sharing without any explicit communications between threads.

The basic idea of the experiment using PThread is similar to that using MPI. After some program initializations, we calculate the problem distribution for each thread. The only thing that is different from PThread is we need to create a function that will act as the routine function of each thread. So, we just need to move the whole Mandelbrot set calculations into that new function with the id of the thread as its parameter. The function of the thread id is to let each thread operates according to its assigned job distribution because each thread will execute the same function. Then, in this new function we can start the calculations by creating loop as the input. After some calculations, the result will be put into a pre-declared global array. Since PThread is a shared-memory based parallelization, it is very important to make sure that there is no data dependency among threads. However, since we have distributed the job among threads, we do not need to worry about the possibility of threads accessing other threads' array address. Then, after all threads have finished their assigned jobs, we can stop the time measurement (that is started before the job distribution) and once again visualize the array using any graphical function.


Program Initialization
PThread Create
Thread 1
Thread 2
Thread 3
Computation
Computation
Computation
PThread Join
Global Array
Visualize Result

Code Instructions

Compilation instructions

Sequential without GUI

```
g++ sequential.cpp -o seq -O2 -std=c++11
```

Sequential with GUI

```
g++ sequential.cpp -o seqg -I/usr/include -L/usr/local/lib -L/usr/lib -lglut -lGLU -lGL -lm -DGUI -O2 -std=c++11
```

MPI without GUI

```
mpic++ mpi.cpp -o mpi -std=c++11
```

MPI with GUI

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

pthread without GUI

```
g++ pthread.cpp -o pthread -lpthread -O2 -std=c++11
```

pthread with GUI

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

To run the sequential program, follow the format:

```
./$PROGRAM $X_RESN $Y_RESN $max_iteration
```

$PROGRAM denotes the executable file you wish to run. $X_RESN and $Y_RESN denote the desired X and Y resolution respectively. $max_iteration is the desired number of iterations for the recurrence relation.

To run the mpi program, follow the format:

```
mpirun -np $n_proc ./$PROGRAM $X_RESN $Y_RESN $max_iteration
```

where $n_proc denotes the number of processors you wish to use.

To run the pthread program, follow the format:

```
./$PROGRAM $X_RESN $Y_RESN $max_iteration $n_thd
```

Where $n_thd denotes the number of threads you wish to use.

If the execution arguments are missing, $X_RESN and $Y_RESN are set default to be 1000 and $n_proc and $n_thd are set to 4 to be their default.

III. Data and Analysis

The experiment is done on the HPC cluster with slurm. Each task is assigned a CPU chip. The program is run with different resolutions as its input size. The max_iteration value is set constant as 100 in all the experiments. The results are as follows:

Processor/Size	*100x100*	*500x500*	*1000x1000*	*5000x5000*
1 (Sequential)	0.004277	0.100951	0.308716	7.948360
4	0.003683	0.081889	0.141824	3.636760
20	0.002036	0.039088	0.040339	0.938730
40	0.001255	0.016273	0.021903	0.709267

Table 1: Execution time for MPI

Threads/Size	*100x100*	*500x500*	*1000x1000*	*5000x5000*
1 (Sequential)	0.009242	0.094030	0.320629	7.628069
4	0.004308	0.053916	0.160940	3.516181
20	0.001842	0.025171	0.065362	0.998503
40	0.001157	0.017971	0.052514	0.694005

Table 2: Execution time for PThread

From looking at the data in both Table 1 and Table 2, we can tell that both the MPI implementation and Pthread implementation decrease in execution time as more processors/threads are added. The sequential program runs similarly fast as the parallel program in small data sizes and numbers, but quickly loses out as the size grows larger. This may happen because the data size is too small, and so, the initialization of the pthread and MPI takes a quite large portion of the overall time. Moreover, more processors mean that pthread and MPI may need more time to initialize each thread/process. Thus, both pthread and MPI lines increase following the increment in threads/processes number.

Comparing the performances of the MPI and Pthread implementations, we can see that both programs are fairly competitive in their execution times in small to medium input data sizes. They have very small – almost negligible – time differences in their execution speed. However, When their processor or thread numbers are still small, we can see that the MPI program is faster than the Pthread program. However, the two programs start becoming competitive again at this data range when their processor/thread number reaches a higher number. It may not be seen with just 5000 x 5000 input size, but pthread could possible overtake MPI on higher input sizes. A possible theory is that the MPI program computes the data elements faster than the Pthread program as it dedicates whole processes instead of just threads (segments of a process). However, as the number of processes grow, so does the communication overhead that MPI requires in the communication between the processes (for scattering and gathering data). This drawback causes the Pthread threads to catch up in execution speed as it has much communication speed when accessing data thanks to its shared data model. This factor lets the Pthread program compete with MPI again, ultimately causing the Pthread program to overcome MPI in execution speed as the thread/processor number grows.

IV. Conclusion

Nowadays, despite the rapid development of computer performance, human still has many problems that still require faster computer. We solve this problem by implementing parallel computing to solve those problems. Thus, this paper examined two parallel computing methods, namely pthread and MPI. We compared those methods by solving Mandelbrot Set problem. After some experiments, we found that both methods have their own advantages and disadvantages. For execution with smaller number of threads/processes, it would be better to use pthread as it is much lighter, more stable, and runs relatively faster with fewer threads. However, if we are looking for parallelization method in a larger scale (for example solving problems with many processes), we should use MPI as it theoretically has no limitation in the number of processes. Besides, we also found that we can improve the performance of pthread by dynamically distributing the problems to all threads. Finally, considering the importance of the parallel computing, further research on this topic is highly valuable.