CSC 4005 Project 2
Mandelbrot Set

Project Introduction

Human nowadays still faces numerous challenges, ranging from issues in pure mathematics to more concrete issues like economic policies. Fortunately, the development of computers in this era of technology has allowed for the solution of many issues, particularly those that may require the use of brute force. Although computer performance is improving quickly every year, some issues could still require better performance to be handled. In order to solve this issue, parallel computing is implemented. Humans developed a number of methods that may provide a better manner of parallelization during the development of parallel computing.

In this programming assignment, we must compute and produce the Mandelbrot set using a variety of parallel programming techniques. The next step is to compare and evaluate the various programs' performance and execution times. The kernel version used is **3.10.0-1160.76.1.el7.x86_64**

Basic Understanding

The Mandelbrot Set is a fascinating occurrence in the fields of complex numbers, number theory, and fractals. The quadratic recurrence equation, where z0=C and C are points in the complex plane, is used to calculate the Mandelbrot set. These complex numbers, where zn's orbit is stable and does not tend to infinity, are included in the Mandelbrot set. The structure and format of the assignment include instructions for computing and using complex numbers mathematically. For the performance study to be as fair and impartial as feasible, all the various programs will adhere to a comparable data flow. The data will be produced based on the desired Mandelbrot Set image resolution. The method will process and compute each pixel in the image to determine whether it is a part of the Mandelbrot set. Each pixel in the image represents a point on the complex plane. The programs will only be evaluated based on their execution times in order to remove noise and other undesired elements. The analysis will not include additional elements like data initialization or GUI creation.

Each program has a max iteration parameter that determines when to instruct it to stop running the recurrence computation. If the point is stable and does not increase significantly (does not tend to infinity) after the program terminates its recursion, it is considered to be a part of the Mandelbrot set and will be coloured black as a result. If not, the point will be rendered in white, producing the fractal image.

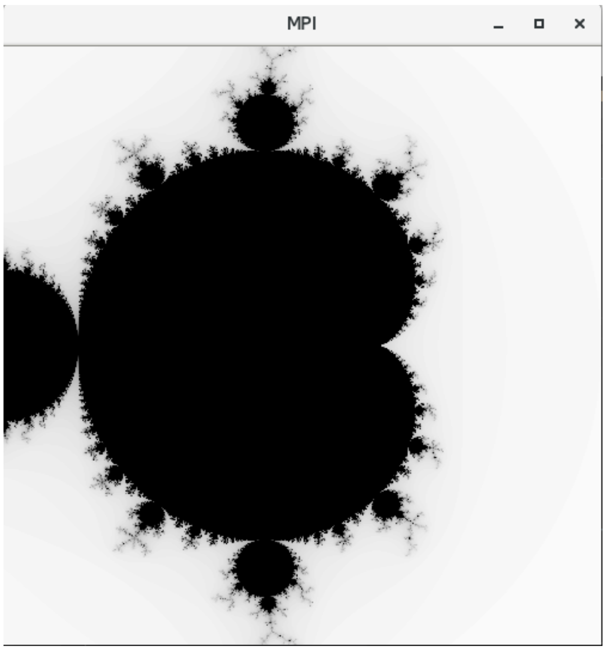


Figure 1. The Mandelbrot GUI program

Mathematical Computation:

$$Z_{k+1} = Z_k^2 + c$$

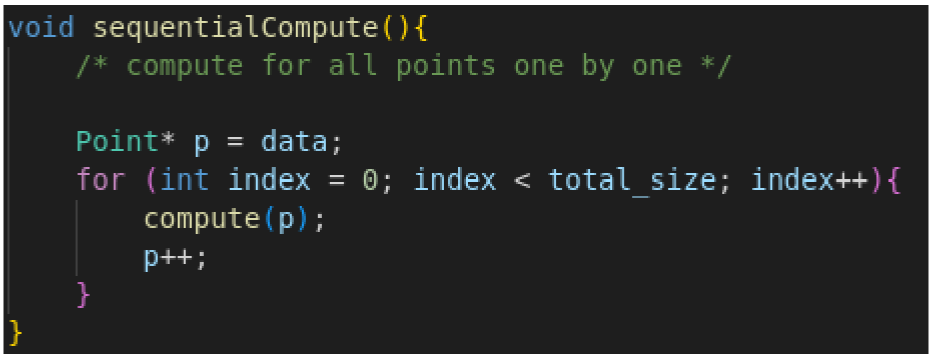
$$Z_k = \sqrt{a^2 + b^2}$$

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

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

Sequential Implementation:

The assignment template and structure are provided along with the sequential program. It is quite simple to put into practice. The data is first initialized by the application using the maximum iteration value and the desired resolution. Then, a for loop is used to determine whether each pixel occurs more than once and to colour it black or white accordingly. The coordinates of each pixel in the complex plane and its colour are saved in an array of a bespoke data structure, which also stores the pixels' data generation.



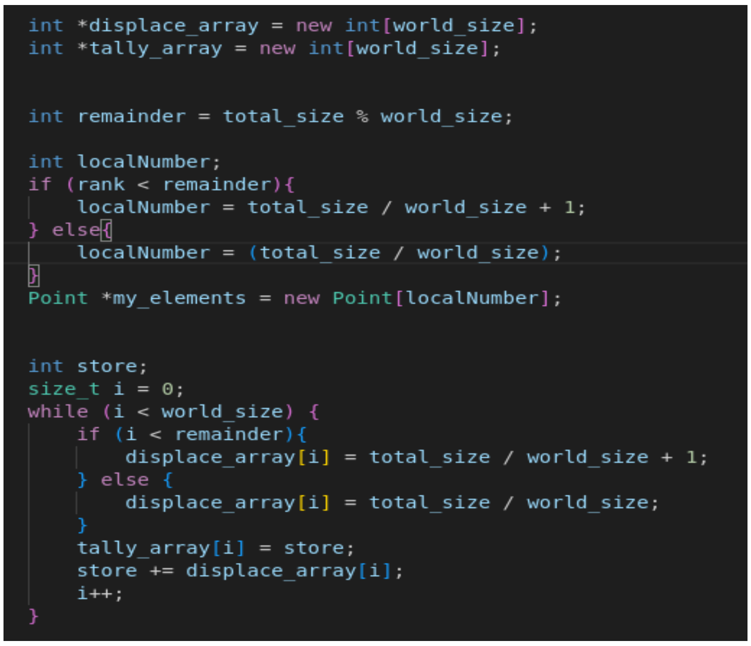
```
void sequentialCompute(){
    /* compute for all points one by one */

    Point* p = data;
    for (int index = 0; index < total_size; index++){
        compute(p);
        p++;
    }
}
```

Figure 2. Given Sequential Computing Code

MPI Implementation:

The Message Passing Interface is used in the initial parallel implementation (MPI). Of this technique, a program is parallelized and computed simultaneously across several processors in a CPU or cluster. Programmers must directly communicate with each processor in order to facilitate data-sharing because MPI lacks shared memory between each unit.

The sequential program and the MPI program have similar logic. Before performing any calculations, the master process initializes the data and distributes it evenly (if possible) to all of the other processes. To avoid wasting RAM, data initialization only takes place at the master process. The algorithm will distribute one of the remaining data elements to each of the other processes, starting with the master, until there are no more remaining (allocates (N% num proc + 1) data. elements to the first (N% num_proc), in the event that the input array size is not divisible by the number of processes. This distribution technique guarantees that any process's input data deviate by no more than one element.



```
int *displace_array = new int[world_size];
int *tally_array = new int[world_size];


int remainder = total_size % world_size;

int localNumber;
if (rank < remainder){
    localNumber = total_size / world_size + 1;
} else{
    localNumber = (total_size / world_size);
}
Point *my_elements = new Point[localNumber];


int store;
size_t i = 0;
while (i < world_size) {
    if (i < remainder){
        displace_array[i] = total_size / world_size + 1;
    } else {
        displace_array[i] = total_size / world_size;
    }
    tally_array[i] = store;
    store += displace_array[i];
    i++;
}
```

Figure 3. MPI Code Implementation

It should be noted that MPI does not by default handle sending and dispersing unique data structs written in C or C++. Therefore, before dispersing the data, a new MPI datatype needs to be defined and committed. This is accomplished by informing MPI of the number of data items in the struct, the data types of each data element, and the amount of memory that each data element requires. The new data type is then created by MPI and committed to, enabling its use all across the program.

Each process's local array serves as the receiving buffer. The data elements from the input that each process will need to compute will be stored in this array. The Mandelbrot set is computed by the MPI software in a manner similar to that of the sequential implementation. The input data array is parallelized, the sole difference being that each processor processes much fewer data elements than a sequential one. The recurrence relation of the data in each process' local arrays will be calculated (computation method is practically the same as the sequential one). The master process will then collect all additional data pieces from the other processors after the calculation and re-arrange them in the data array. The software will then generate the Mandelbrot set image and print the execution time. Due to a problem, the MPI program will in this implementation create many picture windows, only one of which will contain the Mandelbrot set image and the others will all be completely black.

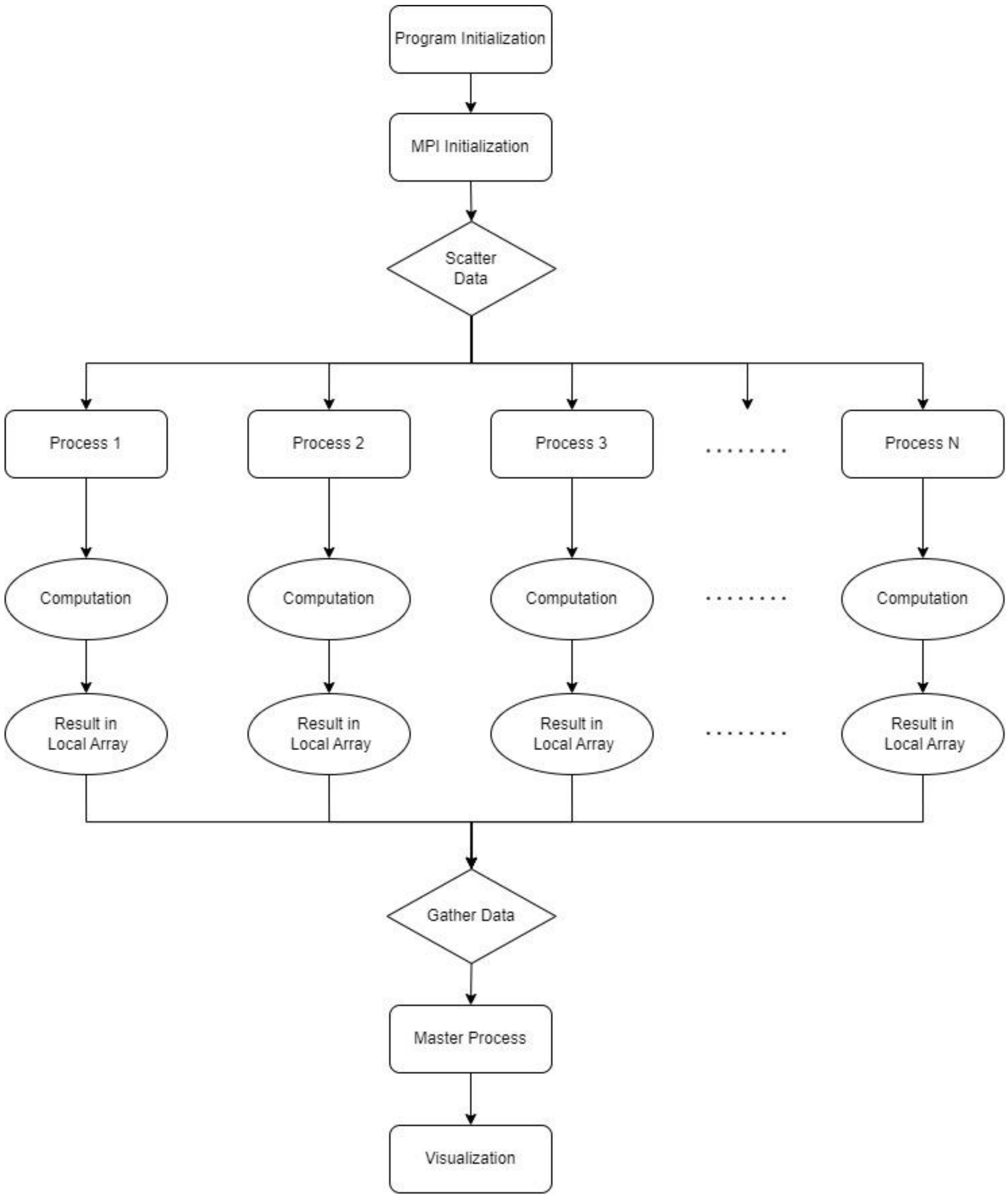


Figure 4. MPI Flow Diagram Implementation

Pthread Implementation:

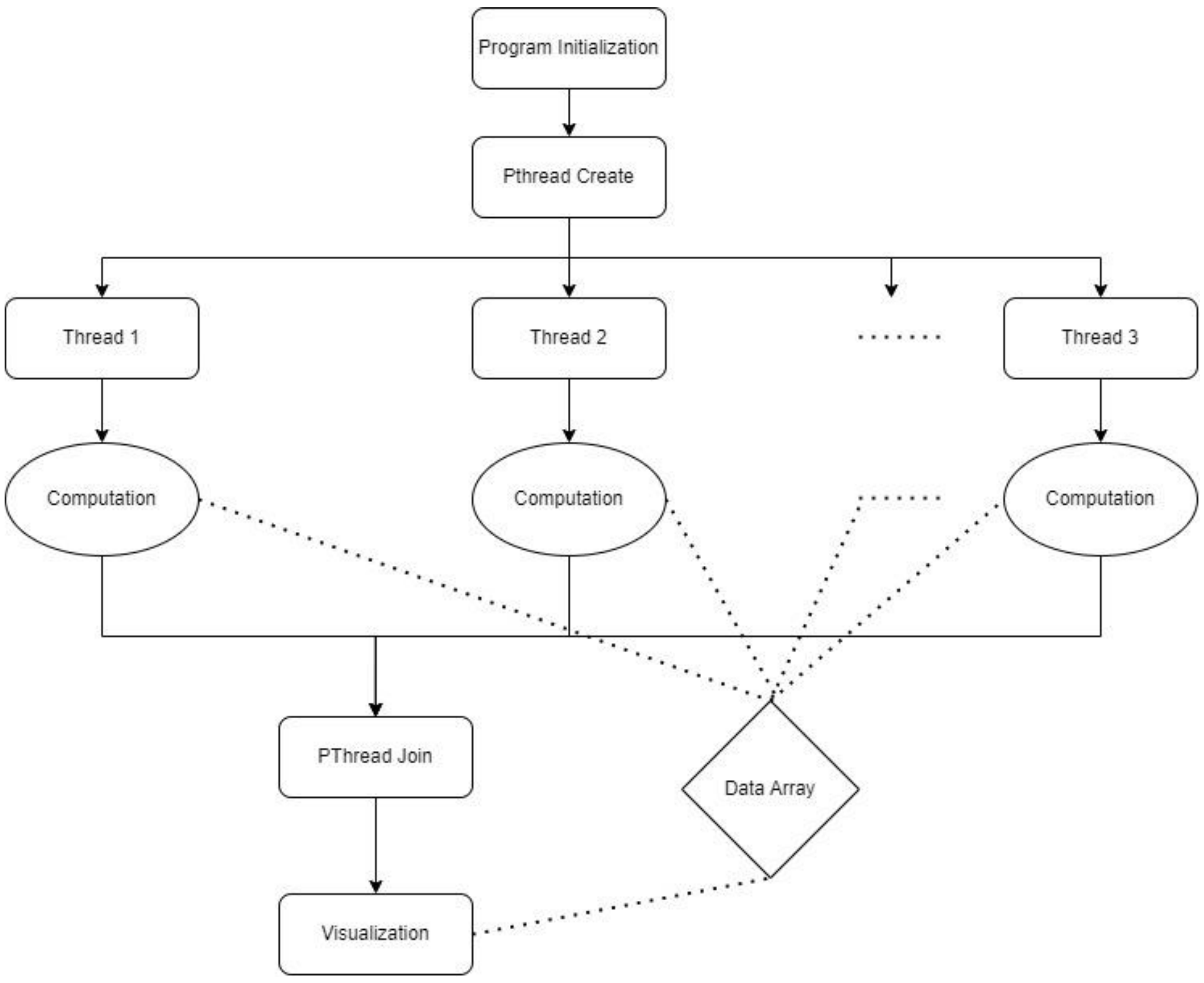


Figure 5. Pthread Flow Diagram Implementation

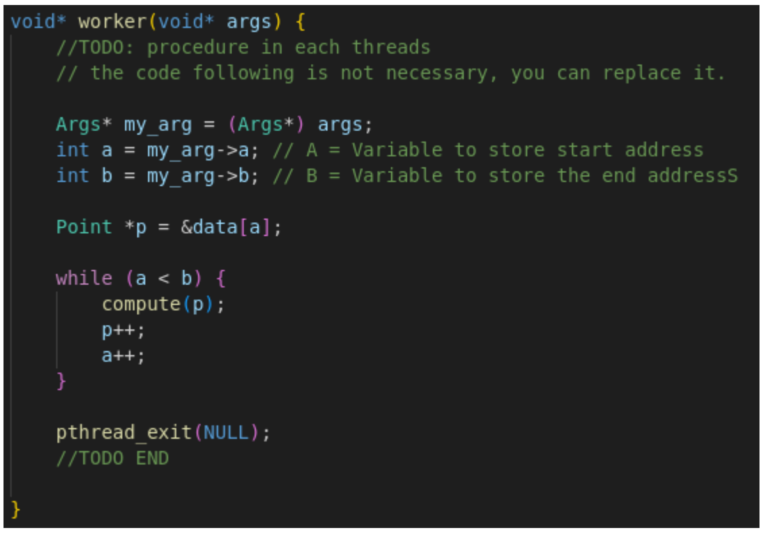
The threads that are present in the computer processor are used in this form of parallelization. This approach, unlike MPI, can be used directly on a personal computer without the requirement for a cluster. Additionally, this system uses threads that are built on shared memory structures. This implies that all threads may share variables, allowing for direct memory sharing without the need for explicit thread-to-thread communication.

The POSIX Threads (Pthread) execution mechanism is utilized in the second parallel implementation. Threads are used in this parallelization technique. In contrast to MPI, which produces and dedicates entire processes to the job, Pthread constructs threads that are portions of processes. Consequently, a process may contain a number of threads. Context switching is significantly simpler because threads still only use one processor and share memory. This method uses a single processor with several threads to run the pthread program.

The logic and data flow used in the MPI implementation are also used in the implementation of the pthread application. In order to spread the data across the threads as evenly as possible, it also contains a remainder system. The arguments given to the threads by the pthread implementation are also stored in a unique data structure. The starting and ending points of the data items in the initialized data array are contained in them. These indicate the beginning (a) and ending (b) points of each thread's computation.

We do not need to split the data array into separate chunks and transmit them to each thread because this programming style uses shared memory across the threads. Sending just the beginning and ending indices of the data elements that they will compute from the original data array is sufficient.

In order for the threads to calculate their respective data pieces, the software will give them a routine function. Following the sequential and MPI implementations is the computation of the recurrence relation. There is no need to collect the data because it is already being processed (within the data array). The Mandelbrot set's image will then be shown by the application.



```
void* worker(void* args) {
    //TODO: procedure in each threads
    // the code following is not necessary, you can replace it.

    Args* my_arg = (Args*) args;
    int a = my_arg->a; // A = Variable to store start address
    int b = my_arg->b; // B = Variable to store the end addressS

    Point *p = &data[a];

    while (a < b) {
        compute(p);
        p++;
        a++;
    }

    pthread_exit(NULL);
    //TODO END

}
```

Figure 6. Variable to store the starting and ending address of the computation

Data Visualization and Analysis

The experiment is carried out using slurm on the HPC cluster. A CPU chip is given to each task. Different resolutions are used as the program's input sizes. In all of the studies, the max iteration variable is kept constant at 100.

Use the format, ./$PROGRAM to run the sequential program. $X_RESN $Y_RESN $max_iteration. The executable file you want to start is indicated by $PROGRAM. The intended X and Y resolutions are indicated by the variables $X_RESN and $Y_RESN. The intended number of iterations for the recurrence relation is $max iteration. Employ the syntax mpirun -np $n_proc ./$PROGRAM $X_RESN $Y_RESN $max iteration to run the mpi program, where $n proc stands for the number of processors you want to use.

Processor Size	100 x 100	500 x 500	1000 x 1000	5000 x 5000	10000 x 10000
1	0.008987s	0.091021s	0.301836s	7.601616s	30.553290s
3	0.004084s	0.049812s	0.176923s	4.6070132s	18.549529s
7	0.002526s	0.029507s	0.087738s	2.105382s	8.315128s
15	0.002189s	0.020380s	0.056308s	1.133518s	4.588232s
30	0.001518s	0.011631s	0.028715s	0.666283s	2.587535s

Table 1. Sequential and MPI Computing Result

Follow this syntax to run the pthread program. /$PROGRAM $X_RESN $Y_RESN $max iteration $n_thd where the number of threads you want to employ is indicated by $n_thd. If the execution arguments are absent, the default values for $X_RESN, $Y_RESN, $n_proc, and $n_thd are set to 1000 and 4, respectively.

Processor Size	100 x 100	500 x 500	1000 x 1000	5000 x 5000	10000 x 10000
1	0.008987s	0.091021s	0.301836s	7.601616s	30.553290s
3	0.005225s	0.058019s	0.179131s	4.382661s	17.251563s
7	0.002868s	0.036440s	0.102446s	2.048859s	8.255839s
15	0.002017s	0.026090s	0.060509s	1.157750s	4.543535s
30	0.002304s	0.018879s	0.0506033s	0.652259s	2.559158s

Table 2. Pthread Computing Result

We can infer that the MPI implementation and the Pthread implementation both see a reduction in execution time as more processors and threads are added by looking at the data in Tables 1 and 2, as well as Figures 1 and 2. It is interesting to notice that the sequential program runs just as quickly—and occasionally faster—than the Pthread and MPI implementations when there are few processes and little data to process. Such a situation can be seen in Table 2 with data input size 500x500 for three threads. The additional time needed to initialize the threads and processes is the cause of the timing discrepancy. But when the number of threads and processes increases, this pattern does not. This can be hypothesized since the Mandelbrot set's processing requires more compute than sorting does. Thus, even at the smallest resolution sizes, the advantage of parallelized computation is readily apparent.

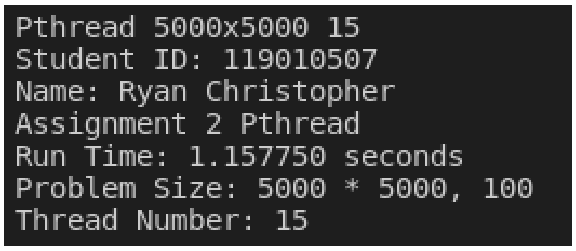
By concentrating on the medium and bigger data ranges, we can observe that the pattern of decreasing execution time with more processes and threads is still present. Figures 1 and 2 demonstrate that the data input size breakpoint, where the parallel algorithms significantly outperform the sequential ones in terms of execution speed, is at 1000x1000.

If we compare the MPI and Pthread implementations' execution speeds for small to medium input data sizes, we can find that both programs are fairly comparable. The difference in the amount of time it takes for them to execute is quite minimal, nearly non-existent. However, we can see that initially, MPI has the advantage when we compare their execution durations in the case of the biggest input size, which is 10000x10000. We can observe that the MPI program is quicker than the Pthread program while the number of their processors or threads is still low. When their respective processor/thread counts hit 15 and 30, the two programs, however, start to become competitive once more at this data range. Their execution time starts to increase once more. After that, Pthread replaces MPI as the quicker program. Given that MPI dedicates entire processes rather than just threads, we can speculate that it computes the data elements more quickly than Pthread (segments of a process). However, as the number of processes increases, so does the amount of communication overhead that MPI mandates for the processes to communicate with one another (for scattering and gathering data). Due to its shared data model and faster communication speeds when accessing data, Pthread threads are able to catch up in terms of execution performance. This allows Pthread to once more compete with MPI, and when the number of threads and processors increases past 15, Pthread eventually outperforms MPI in terms of execution performance.

Project Summary

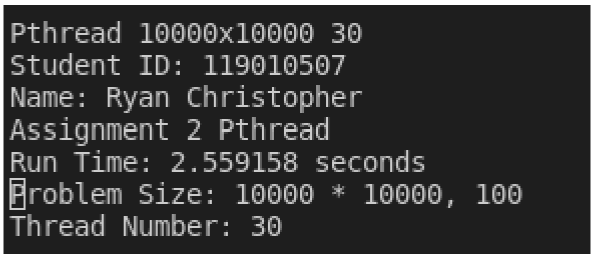
This project helped us understand the differences in running time between MPI and Pthread, among other parallel computing architectures. When compared to sequential programs, MPI and Pthread both parallelize execution and speed up performance with huge data inputs. However, MPI increases the number of processors whereas Pthread increases the number of threads due to their differing execution methods. Due to this, MPI lacks a shared data memory model, but Pthread does. We can observe from the Mandelbrot set that for the majority of input data sizes, the MPI program and Pthread have competitive execution times. We can see that MPI computes quicker than Pthread but has higher overhead when the input sizes are large enough. Consequently, Pthread scales with additional threads better than MPI. Therefore, it is crucial to understand the context of the program's running environment and how many processors or threads to employ to achieve the quickest execution time when choosing which execution model to use.

Project Sample Output



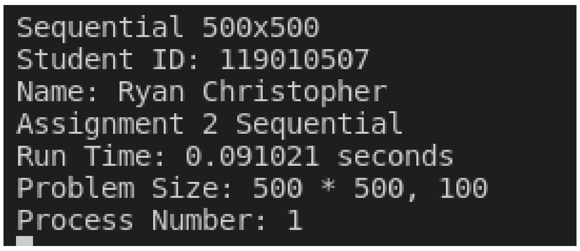
```
Pthread 5000x5000 15
Student ID: 119010507
Name: Ryan Christopher
Assignment 2 Pthread
Run Time: 1.157750 seconds
Problem Size: 5000 * 5000, 100
Thread Number: 15
```

Figure 7. Pthread Sample 5000x5000 15



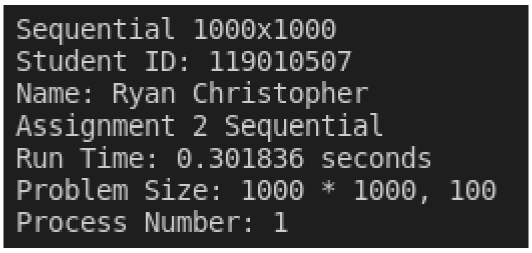
```
Pthread 10000x10000 30
Student ID: 119010507
Name: Ryan Christopher
Assignment 2 Pthread
Run Time: 2.559158 seconds
Problem Size: 10000 * 10000, 100
Thread Number: 30
```

Figure 8. Pthread Sample 10000x10000 30



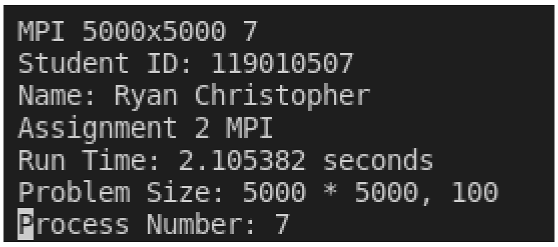
```
Sequential 500x500
Student ID: 119010507
Name: Ryan Christopher
Assignment 2 Sequential
Run Time: 0.091021 seconds
Problem Size: 500 * 500, 100
Process Number: 1
```

Figure 9. Sequential Sample 500x500 1



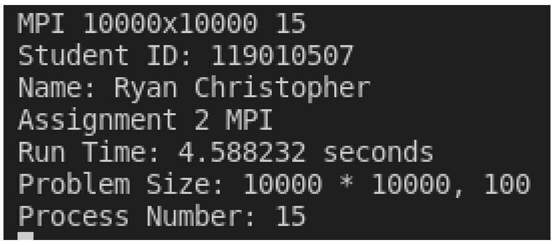
```
Sequential 1000x1000
Student ID: 119010507
Name: Ryan Christopher
Assignment 2 Sequential
Run Time: 0.301836 seconds
Problem Size: 1000 * 1000, 100
Process Number: 1
```

Figure 10. Sequential 1000x1000 1



```
MPI 5000x5000 7
Student ID: 119010507
Name: Ryan Christopher
Assignment 2 MPI
Run Time: 2.105382 seconds
Problem Size: 5000 * 5000, 100
Process Number: 7
```

Figure 11. MPI Sample 5000x5000 7



```
MPI 10000x10000 15
Student ID: 119010507
Name: Ryan Christopher
Assignment 2 MPI
Run Time: 4.588232 seconds
Problem Size: 10000 * 10000, 100
Process Number: 15
```

Figure 12. MPI Sample 10000x10000 15

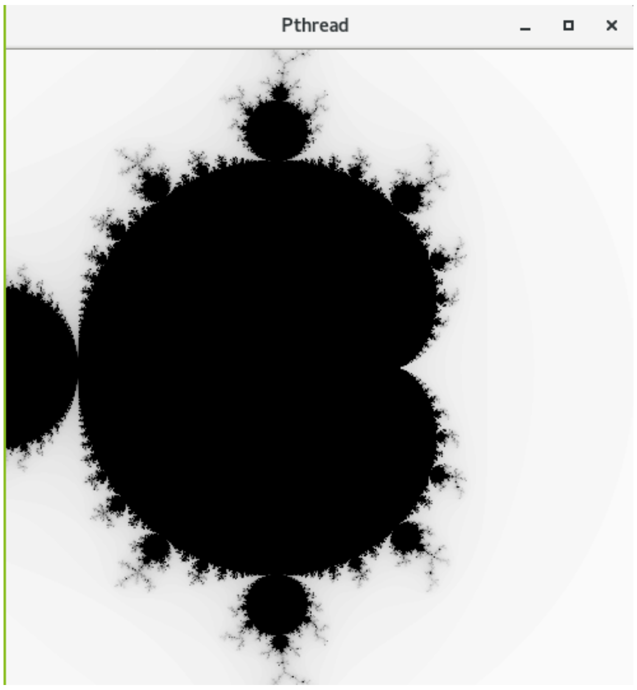


Figure 13. Pthread Sample GUI