The Chinese University of Hong Kong, Shenzhen

CSC4005

Parallel Programming

Project 2 Report

Author:
Yang Hongmeng

Student Number:
119010378

October 31, 2022

Contents

1 Introduction

The Mandelbrot set is the set of values of c in complex plane for which the orbit of the critical point z=0 under iteration of the quadratic map: $z_(n+1) = (z_n)^2 + c$ remains bounded. Thus, a complex number c is a member of Mandelbrot set if, when starting with z˙0 = 0 and applying the iteration repeatedly, the absolute value of z˙n remains bounded for all n¿0. In order to exploring the Mandelbrot Set Computation, I use both MPI and Pthread methods to implement the computation method and get the figure of computation result.

MPI method utilizes multiple processes to accomplish the parallel computing, and Pthread method uses multiple threads to do the parallel computing. Each process can have multiple threads and multiple threads in a process can share the memory. Therefore, for some cases multi-threads implementation could not have too much communication time compared with multi-processes implementation.

In this project, we can compare the difference between two kinds of methods of implementation. At the same time, the different execution time can also show some differences between two implementation.

According to the given template, we do not need to implement the algorithm of Mandelbrot Set Computation. The main part of this project is how to segment the input data to different processes or threads. There are also some special cases we need to take care of. For example, if the total number of elements which should be computed cannot be divisible by the process number or thread number, how to separate the input element to different processes and threads.

The detailed implementation methods and explanation are shown below.

2 Material and Method

As introduced above, the main part of this project is how to segment the input data into different processes and threads. Therefore, I will introduce how I implement the segmentation function of different two methods. At the same time, I will also show how to execute my program in this part.

The first step of implement the parallel computing version of this program is to understand the sequential version.

From the sequential version, we know that we need to pass a “Point” pointer to the function of “compute()” and this function can help us to compute the result of the input Point matrix directly. At the same time, we can also know that the process

which call the function of "initData()" can directly get the input data by utilizing a Point pointer.

After getting the computational result, whether we need to output the image depends on the compilation method. When we executing the program, the execution time related to certain execution would also showed in the output part.

2.1 MPI computation method

2.1.1 Theoretical Model

If we want to change the sequential version of algorithm to MPI method, the most important part is how to separate the input "Point" array to each process and how to let processes to communicate with each other.

According to the template, it is obvious that the master process is process 0 which is used to segment the input array to all other processes and receive the calculation results from all those processes. Therefore, I design the "master()" function do segmentation. At the same time, the other processes just need to receive data and do calculation. After calculation, they send all result back to process 0. The detail implementation are shown below.

First, before function "master()" separates input data, it should have access to the initialized input data. The initialized data is a global variable named "data", and I use another "Point" pointer to get the data from "data". At the same time, I use an integer to store the total size of input data named "Total_size". In order to do further operation to the input data and protect the initial data, I copy the initialized data to a vector.

Second, segment data to each process. In order to use the "MPI_Scatter()" function to separate data and send data to each process, I need to make sure the total number of input data can be divisible by the process number. Therefore, I check this at the beginning of second process.

In order to check whether the total size of data is divisible by the number of process, the program would first calculate the remainder of total size of data divides the number of process. If remainder is equal to 0, the program would do nothing to the input data and prepare for scatter data to each process. However, if the remainder is not equal to 0, which means the total size of data is not divisible by the number of process, the program would the last element of the initial data and make sure it would be divisible.

After processing the input data, the master process(process 0) would first broadcast the number of element each process should receive. After getting this number, each

process would create a local vector of "Point" to store the data received from master process. At this time, all process are ready to receive data.

The next step is that the master process scatter initialized data to each process by utilizing the function "MPI_Scatter()". Nevertheless, the new problem is that the data type here is "Point" which I create it by myself. It is not a MPI type. Therefore, we need to create a MPI data type. In my definition, the Point_TYPE is a MPI data type which is same as structure "Point".The integer vector "blocklength" stores the block number of each data in the structure should use, and the MPI_datatype vector "types" stores the MPI_datatype of each element in the structure of "Point". "Displacements" stores the pointer of each element in "Point".

After creating the MPI data type, the program can scatter the "Point" data to each process. All of process will store the data from master process to the local vector. Then each process can compute the result respectively. According to the implementation of function of "compute()", the program just need to put the "Point" pointer to the function then the result would be put into it directly.

Finally, the master process gather the calculation result from each process ans copy it back to the original data array according to the total size of input data.

2.1.2 Detailed implementation

Master process execution

First, get the global variable of input array named "data" to pointer p, and get the total size of input data. In order to protect the original data, I copy the initial data to another vector in this program,which is also easy for further operation. Therefore, the time complexity here is O(n).

```
Point* p=data;
int Total_size = total_size;
std::vector<Point> calculate_Vector;
std::copy(p,p + Total_size,std::back_inserter
(calculate_Vector));
```

Figure 1: Master process get the input data

Second, I calculate the remainder of total size of divided by the number of the number of process. If the remainder is equal to 0, then the program will do nothing. However, if the remainder is not equal to 0, the program will add (process number - remainder) elements into the vector, in order to make sure when the master process

do the scatter function, it could be equally divided. I choose the last element of the initialized data to be the element add to the end of vector.

After processing the data, the program will recalculate the number of data each process would get. Finally, the master process would broadcast this number and all of process would create local vector which is use to store the following data.

```
int calcu_size = Total_size;
int local_data_size = Total_size/world_size;
int remainder = total_size%world_size;
if(remainder != 0){
    int lack_element = world_size - remainder;
    calcu_size += lack_element;
    Point last_ele = calculate_Vector[Total_size-1];
    for(int i = 0; i < lack_element; i++){
        calculate_Vector.push_back(last_ele);
    }
    local_data_size = calcu_size/world_size;
}
MPI_Bcast(&local_data_size,1,MPI_INT,0,MPI_COMM_WORLD);
```

Figure 2: Input data processing

The next step is the same as the other process, master process also need to create a local buffer to store the point the master process should calculate. And then the master process can scatter data to the other processes. After scattering data to all processes, the master process can compute its data.

Finally, the master process gather all data back and copy it to the original input array.

```
std::vector<Point> local_data_buffer(local_data_size);

MPI_Scatter(calculate_Vector.data(), local_data_size,
Point_TYPE, local_data_buffer.data(), local_data_size,
Point_TYPE, 0, MPI_COMM_WORLD);
for(int i = 0; i < local_data_size; i++){
    compute(&local_data_buffer[i]);
}
MPI_Gather(local_data_buffer.data(), local_data_size,
Point_TYPE, calculate_Vector.data(), local_data_size,
Point_TYPE, 0, MPI_COMM_WORLD);
std::copy(calculate_Vector.begin(), calculate_Vector.begin()
+(Total_size), data);
```

Figure 3: Final step of master() function

Other process execution

For those processes except master process, they just need to execute the "slave()" function. In this function, process should get the number of elements each process need to process by utilizing the "MPI_Bacst()" function. After getting the number, the process should create a vector which is used to store data from master process.

```
int local_data_size;
std::vector<Point> calculate_Vector;
MPI_Bcast(&local_data_size,1,MPI_INT,0,MPI_COMM_WORLD);
std::vector<Point> local_data_buffer(local_data_size);
```

Figure 4: Initialization of slave() function

After creating the local buffer, all process would receive the input data from master process by using the function of "MPI_Scatter()". And then all process would calculate its data and finally send the result back to the master process by utilizing the function "MPI_Gather()".

```
MPI_Scatter(calculate_Vector.data(), local_data_size,
Point_TYPE, local_data_buffer.data(), local_data_size,
Point_TYPE, 0, MPI_COMM_WORLD);
for(int i = 0; i < local_data_size; i++){
    compute(&local_data_buffer[i]);
}
MPI_Gather(local_data_buffer.data(), local_data_size,
Point_TYPE, calculate_Vector.data(), local_data_size,
Point_TYPE, 0, MPI_COMM_WORLD);
```

Figure 5: Final step of slave() function

MPI datatype creation

In the following figure, I show how to create a MPI data type which can be used to scatter a user define structure.

```
const int count = 3;
int blocklengths[3] = {1,1,1};
MPI_Datatype types[3] = {MPI_INT, MPI_INT, MPI_FLOAT};
MPI_Aint displacements[3];
displacements[0] = offsetof(Point,x);
displacements[1] = offsetof(Point,y);
displacements[2] = offsetof(Point,color);
MPI_Type_create_struct(count, blocklengths, displacements,
types, &Point_TYPE);
MPI_Type_commit(&Point_TYPE);
```

Figure 6: Creation of MPI datatype

2.2 Pthread computation method

2.2.1 Theoretical Model

First, create n threads by using the function of "pthread˙create()" and it will also pass the data each thread need to all threads. When the program execute the function "pthread˙create()" all threads also execute the function of worker.

Before threads can execute the function of "worker()", all threads should get the number of elements they need to process and they also need to know whether the number of input element is divisible by the number of process. Therefore, in the main function the program would pass a variable named "args" of the structure "Args" to all threads.

The structure "Args" has three elements in it. The first element indict the thread ID, which is helpful for each thread finding elements they need to process. The second element in "Args" is the number of elements each process should calculate. The last element indicts whether the number of input element is divisible by the number of process.

For the function each thread process, it only has two parts: get input data and calculate data. For getting input data, we just need to use a same structure variable to the args input. At the same time, each process should create three local variables to store these data.

The second step is computing the result according to the given remainder. If the remainder is 0, which means the number of input data can be divisible by the number of process. Then each thread just compute from (thread_ID * average_number_of_element) to (thread_ID + 1) * (average_number_of_element). If the remainder is not equal to 0, each thread should get one more data from the global variable "data" and the last thread can get less data than others. However, in order

to to control this process, I use the total size of data as an condition which is used to make sure that the last thread would not exceed the data range.

2.2.2 Detailed implementation

The following figure shows the detail implementation of key function of pthread implementation of this project

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

    Args* my_arg = (Args*) args;
    int a = my_arg->a;
    int average_ele = my_arg->b;
    int remainder = my_arg->c;

    if(remainder == 0){
        for (int i = a*average_ele; i<(a+1)*average_ele;i++){
            compute(data+i);
        }
    }
    else{
        average_ele+=1;
        for (int i = a*average_ele; i<(a+1)*average_ele && i<
        total_size;i++){
            compute(data+i);
        }
    }
}
```

Figure 7: Pthread implementation

2.3 Compile and test both codes

In this section, I will introduce how to compile and test my code.

In order to compile and execute my code, you can directly run file “sequential.sh” , “para_MPI.sh”and “para_Pth.sh” by utilizing command “sh sequential.sh” , “sh para_MPI.sh” or “para_Pth.sh. These three files will help you compile three of my code directly and execute them. If you want to change the test dataset, you can

change the range of variable "ele_num" in "para_MPI.sh" and "para_Pth.sh". You can also change the core number of execution by changing the number of variable of "core" in file "para_MPI.sh" and "para_Pth.sh".

When you execute these three files, they will create three folder to store the result. For parallel execution, you can check result in folder "para_MPIPth_report_result". In this folder you can choose to see different core number execution by choosing the folder name with the core number.

3 Result Analysis:

3.1 Theoretical analysis

3.1.1 Time complexity

Sequential implementation:

For sequential implementation of the algorithm, the first step is initializing data. In this part, the time complexity is equal to O(X_size * Y_size). Because (X_size * Y_size) = total size, the time complexity can also be represented by O(n).

The second step is calculating the result. In this section, the program loop all of point in the array and for each point it would also loop the whole array again. Therefore, it is obvious that the time complexity of calculation part is O(n^2).

Parallel implementation:

MPI implementation:

By using MPI method to implement the algorithm, each process get ($\frac{n}{p}$) elements, and when calculating the result of the elements, each element loop the whole array which time complexity is O(n). Therefore, the time complexity of MPI method of implementation is O($\frac{n^2}{p}$)

Pthread implementation:

When we use Pthread method, the time complexity is the same as the MPI method. Each thread can get at most $\frac{n}{p} + 1$ elements, and when each element execute the "compute()" function, they loop the whole array which time complexity is O(n). Therefore, the time complexity of the Pthread method is also O($\frac{n^2}{p}$)

3.1.2 Speedup

According to the equation of speedup, we know S(n)=$\frac{t_s}{t_p}$. Speedup is equal to the execution time of sequential execution (t_s) divided by execution time of parallel

execution (t_p).

3.1.3 Efficiency

According to the equation of efficiency we know E=$\frac{S(n)}{n}$. Efficiency is equal to the speedup of n processes divided by process number.

3.2 Result analysis

3.2.1 Execution time

Figure below shows the execution time of sequential execution of current calculation algorithm. According to our analysis above, we can know if the input number enlarge to twice of original input, execution would be four times of original execution time. The curve showed on the figure is the execution time of input size of 400 times square of input elements number divide 400. We can conclude from the figure that the experiment results are near the line. Therefore, it can be an evidence of time complexity of sequential execution is O(n^2).

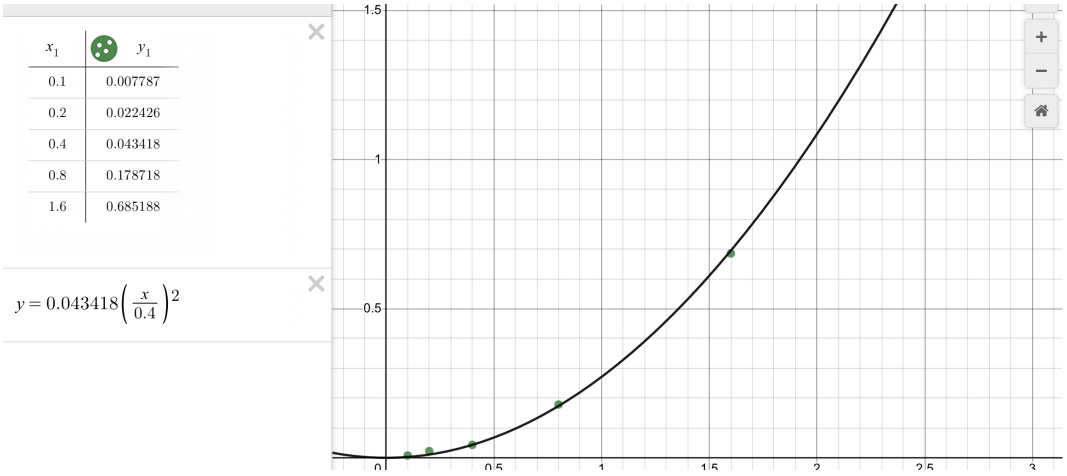
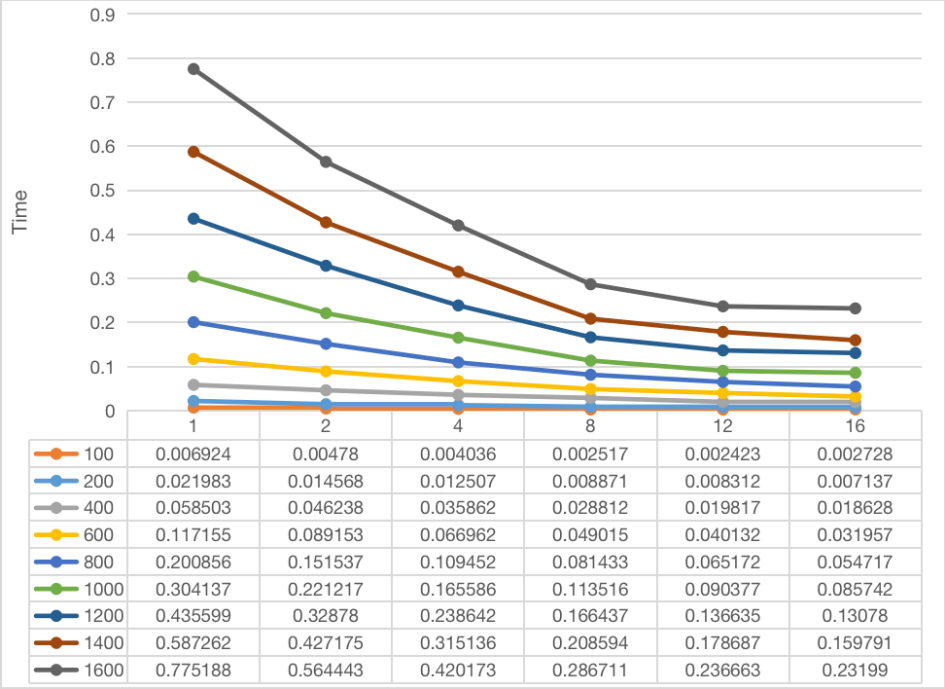


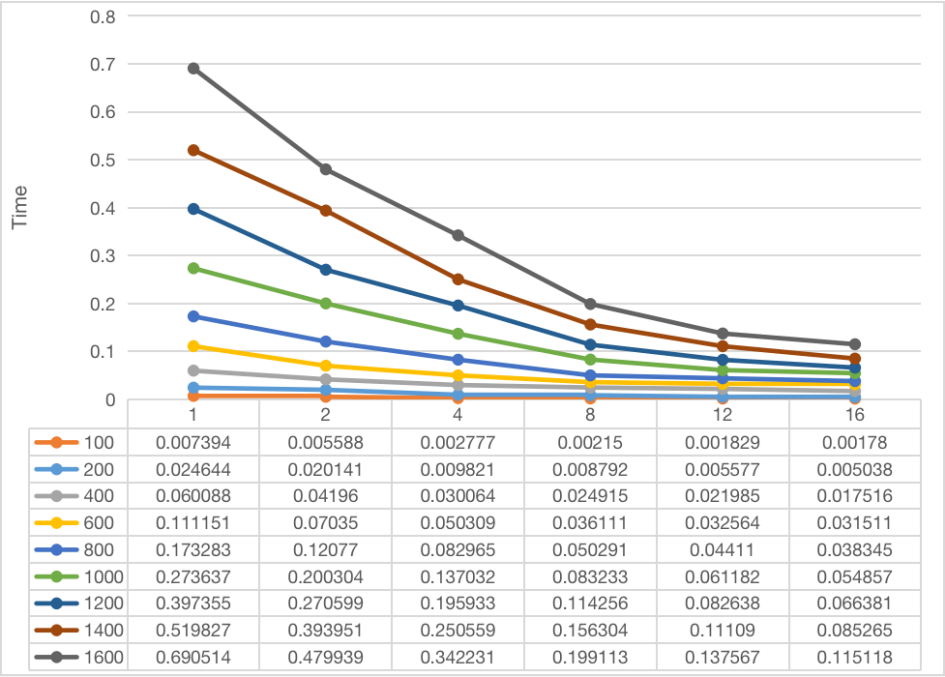
Figure 8: Sequential execution result

We can learn from these two figures is that, with the increase of process core numbers or thread numbers, the execution time decreases. At the same time, when the input data is very small, MPI method is faster than Pthread. However, if the input data is large enough, Pthread is faster. This may be because Pthread utilizing share memory method, which decrease the time of communication compared with MPI method. Because MPI would have lots of inter process communication, the execution time of MPI method is longer is reasonable.



	1	2	4	8	12	16
100	0.006924	0.00478	0.004036	0.002517	0.002423	0.002728
200	0.021983	0.014568	0.012507	0.008871	0.008312	0.007137
400	0.058503	0.046238	0.035862	0.028812	0.019817	0.018628
600	0.117155	0.089153	0.066962	0.049015	0.040132	0.031957
800	0.200856	0.151537	0.109452	0.081433	0.065172	0.054717
1000	0.304137	0.221217	0.165586	0.113516	0.090377	0.085742
1200	0.435599	0.32878	0.238642	0.166437	0.136635	0.13078
1400	0.587262	0.427175	0.315136	0.208594	0.178687	0.159791
1600	0.775188	0.564443	0.420173	0.286711	0.236663	0.23199

Figure 9: Parallel execution result with MPI method



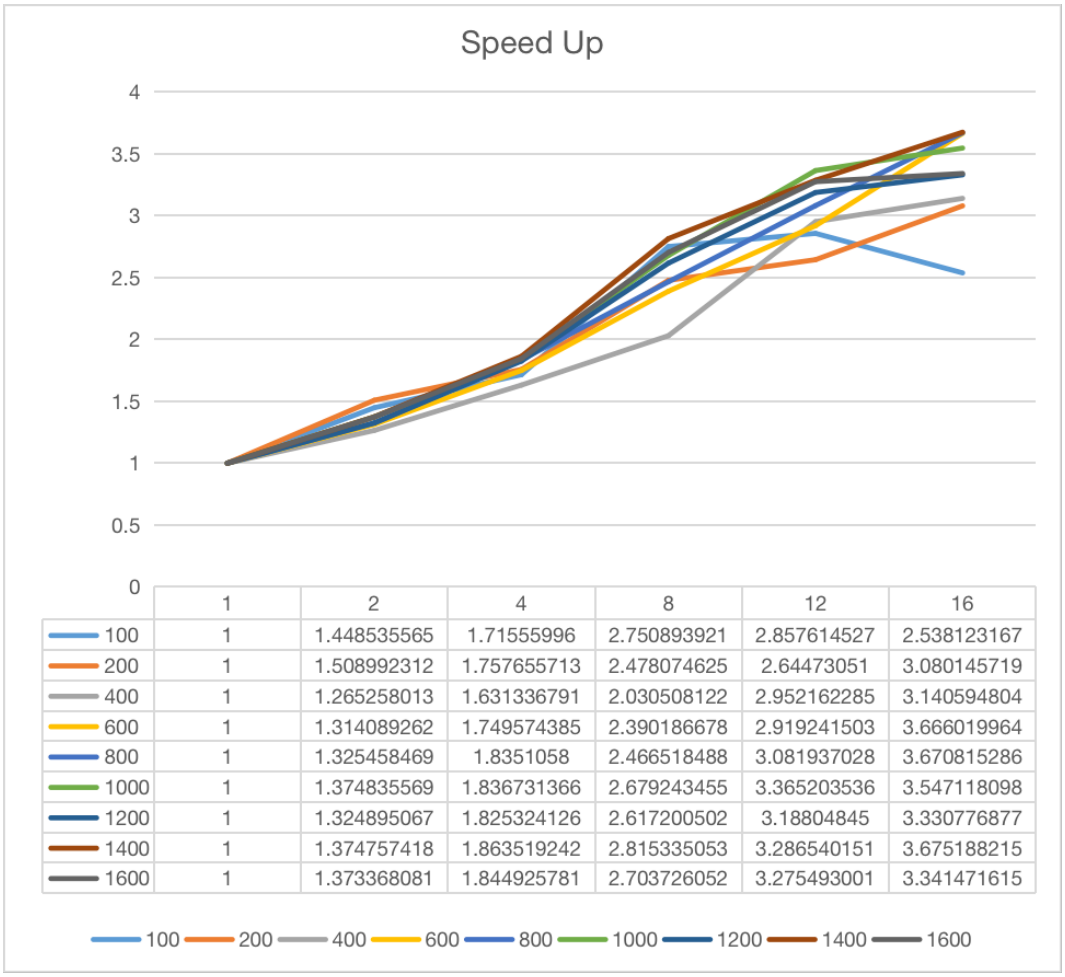
	1	2	4	8	12	16
100	0.007394	0.005588	0.002777	0.00215	0.001829	0.00178
200	0.024644	0.020141	0.009821	0.008792	0.005577	0.005038
400	0.060088	0.04196	0.030064	0.024915	0.021985	0.017516
600	0.111151	0.07035	0.050309	0.036111	0.032564	0.031511
800	0.173283	0.12077	0.082965	0.050291	0.04411	0.038345
1000	0.273637	0.200304	0.137032	0.083233	0.061182	0.054857
1200	0.397355	0.270599	0.195933	0.114256	0.082638	0.066381
1400	0.519827	0.393951	0.250559	0.156304	0.11109	0.085265
1600	0.690514	0.479939	0.342231	0.199113	0.137567	0.115118

Figure 10: Parallel execution result with Pthread method

3.2.2 Speedup

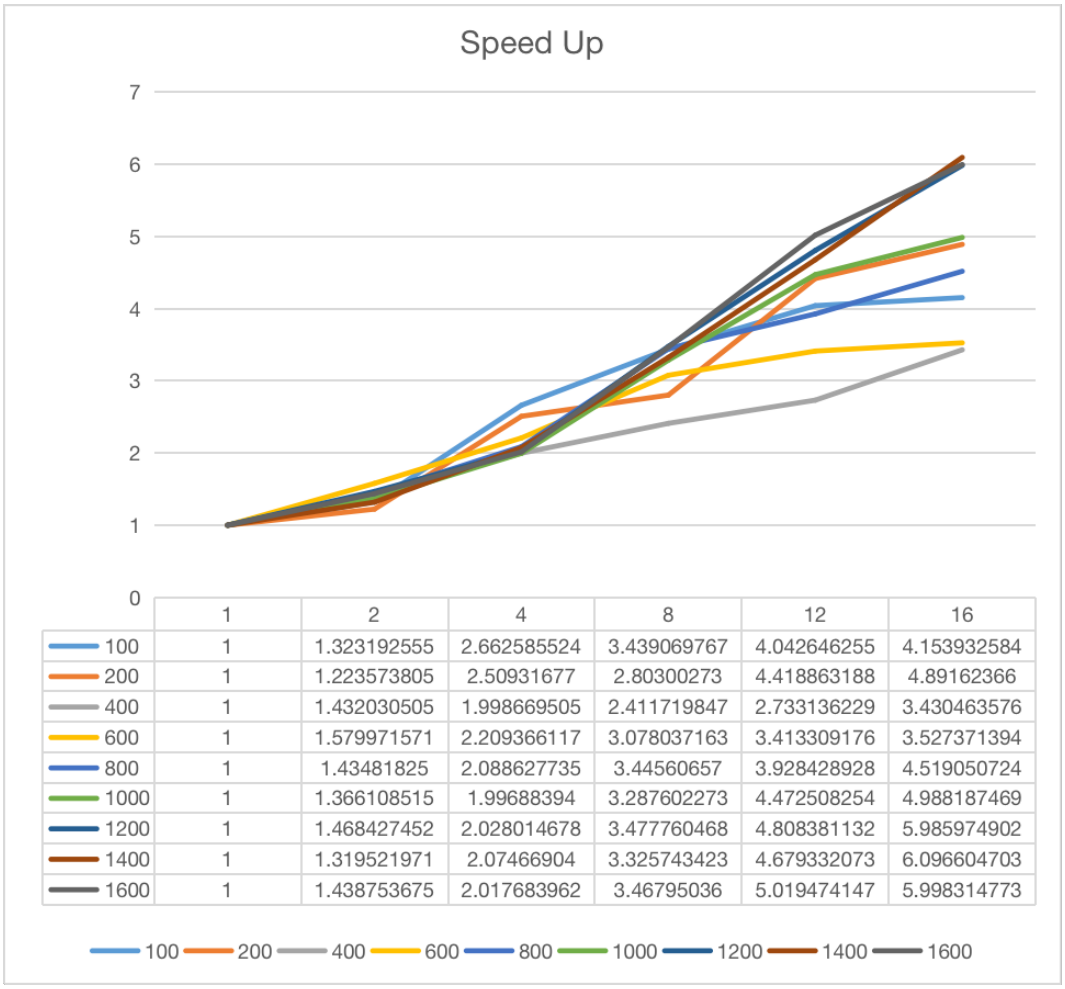
The following figures show the speedup of my test result of two parallel computing methods. From these figure we can know with the increase of process number or thread number, the speedup increases. However, we can also learn that the increasing rate of speedup is not a constant. The possible reason is that this algorithm cannot be finished totally by only parallel executing. There are some communication between each process, and there are also sequential execution in this algorithm.

At the same time, we can also find that the speedup of Pthread implementation is higher than MPI method. The possible reason is that Pthread implementation has no inter thread communication because of share memory among all threads. However, there are lots of communication between processes among processes in the MPI implementation.



	1	2	4	8	12	16
100	1	1.448535565	1.71555996	2.750893921	2.857614527	2.538123167
200	1	1.508992312	1.757655713	2.478074625	2.64473051	3.080145719
400	1	1.265258013	1.631336791	2.030508122	2.952162285	3.140594804
600	1	1.314089262	1.749574385	2.390186678	2.919241503	3.666019964
800	1	1.325458469	1.8351058	2.466518488	3.081937028	3.670815286
1000	1	1.374835569	1.836731366	2.679243455	3.365203536	3.547118098
1200	1	1.324895067	1.825324126	2.617200502	3.18804845	3.330776877
1400	1	1.374757418	1.863519242	2.815335053	3.286540151	3.675188215
1600	1	1.373368081	1.844925781	2.703726052	3.275493001	3.341471615

Figure 11: Speedup of parallel execution with MPI method



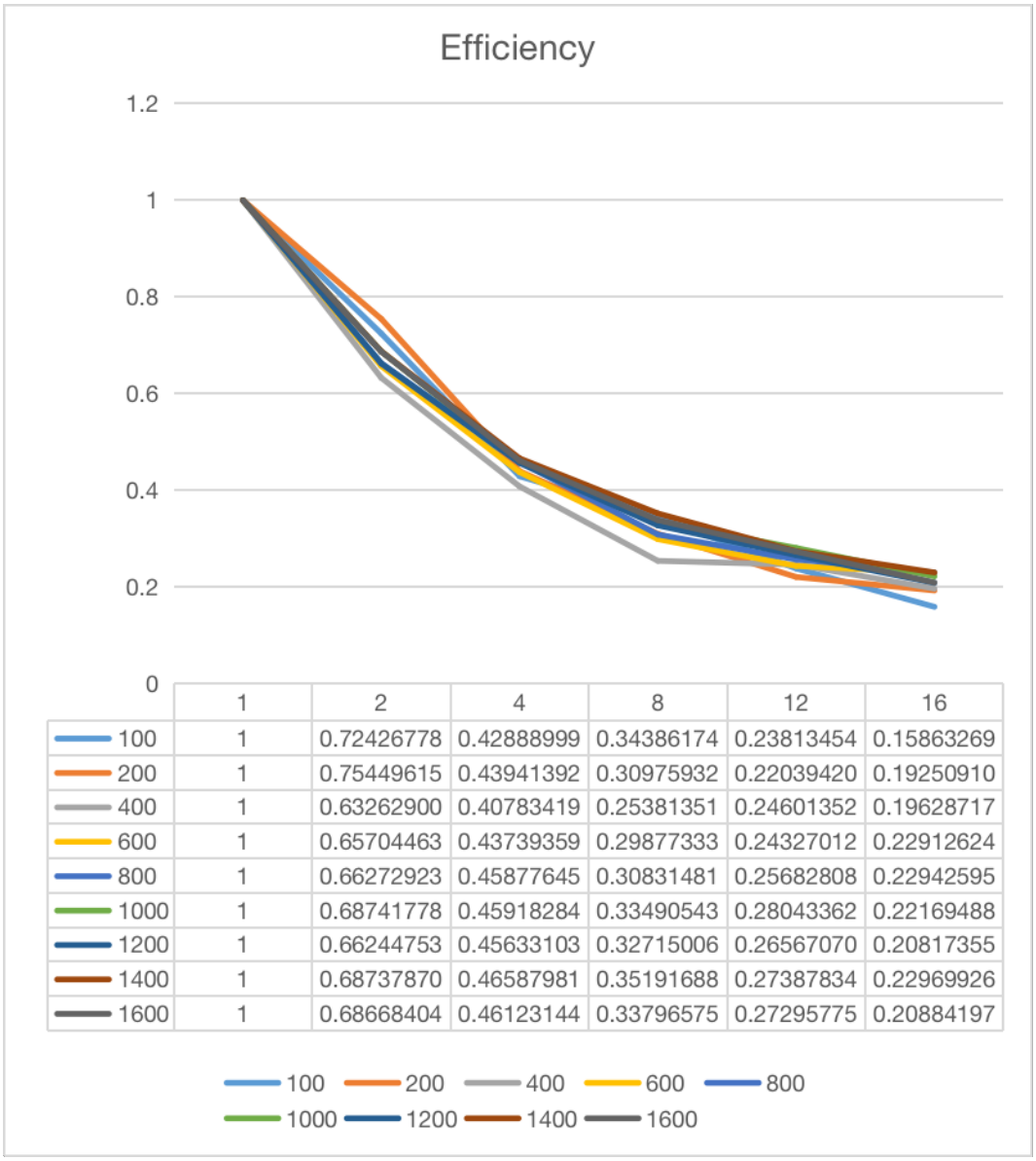
	1	2	4	8	12	16
100	1	1.323192555	2.662585524	3.439069767	4.042646255	4.153932584
200	1	1.223573805	2.50931677	2.80300273	4.418863188	4.89162366
400	1	1.432030505	1.998669505	2.411719847	2.733136229	3.430463576
600	1	1.579971571	2.209366117	3.078037163	3.413309176	3.527371394
800	1	1.43481825	2.088627735	3.44560657	3.928428928	4.519050724
1000	1	1.366108515	1.99688394	3.287602273	4.472508254	4.988187469
1200	1	1.468427452	2.028014678	3.477760468	4.808381132	5.985974902
1400	1	1.319521971	2.07466904	3.325743423	4.679332073	6.096604703
1600	1	1.438753675	2.017683962	3.46795036	5.019474147	5.998314773

Figure 12: Speedup of parallel execution with Pthread method

3.2.3 Efficiency

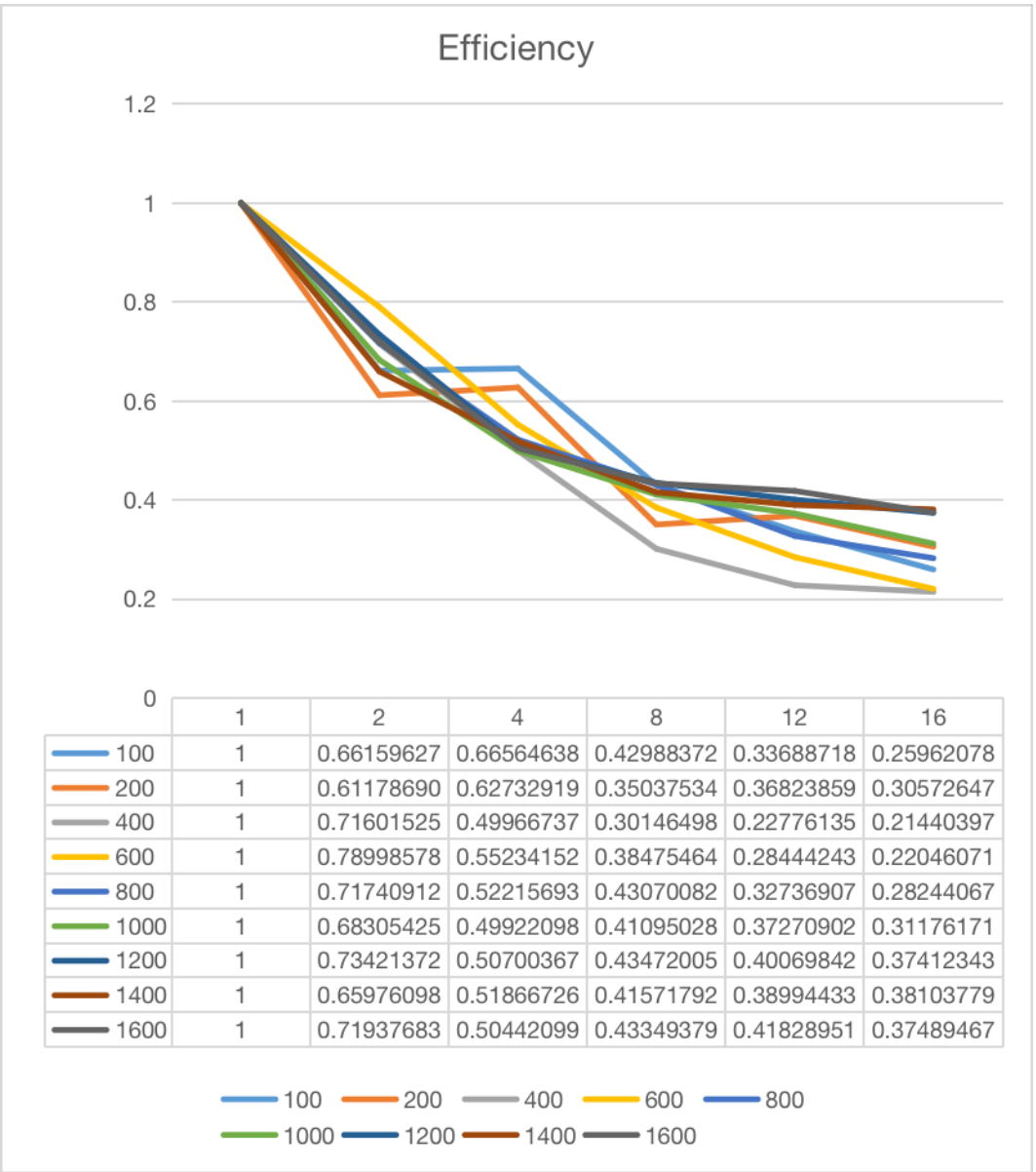
After getting the speedup, I use speedup divide their corresponding process number or thread number to get efficiency of the experiments. And the results are showed in below figures. The efficiency shows descending tendency with the increase of process number. With the increase of input elements number, the efficiency increases.

At the same time, because the speedup of the pthread method is higher than MPI method, the efficiency of pthread is also higher than MPI. Therefore, from this experiment and according to my implementation of these two methods, pthread implementation has higher efficiency.



	1	2	4	8	12	16
100	1	0.72426778	0.42888999	0.34386174	0.23813454	0.15863269
200	1	0.75449615	0.43941392	0.30975932	0.22039420	0.19250910
400	1	0.63262900	0.40783419	0.25381351	0.24601352	0.19628717
600	1	0.65704463	0.43739359	0.29877333	0.24327012	0.22912624
800	1	0.66272923	0.45877645	0.30831481	0.25682808	0.22942595
1000	1	0.68741778	0.45918284	0.33490543	0.28043362	0.22169488
1200	1	0.66244753	0.45633103	0.32715006	0.26567070	0.20817355
1400	1	0.68737870	0.46587981	0.35191688	0.27387834	0.22969926
1600	1	0.68668404	0.46123144	0.33796575	0.27295775	0.20884197

Figure 13: Efficiency of parallel execution with MPI method



	1	2	4	8	12	16
100	1	0.66159627	0.66564638	0.42988372	0.33688718	0.25962078
200	1	0.61178690	0.62732919	0.35037534	0.36823859	0.30572647
400	1	0.71601525	0.49966737	0.30146498	0.22776135	0.21440397
600	1	0.78998578	0.55234152	0.38475464	0.28444243	0.22046071
800	1	0.71740912	0.52215693	0.43070082	0.32736907	0.28244067
1000	1	0.68305425	0.49922098	0.41095028	0.37270902	0.31176171
1200	1	0.73421372	0.50700367	0.43472005	0.40069842	0.37412343
1400	1	0.65976098	0.51866726	0.41571792	0.38994433	0.38103779
1600	1	0.71937683	0.50442099	0.43349379	0.41828951	0.37489467

Figure 14: Efficiency of parallel execution with Pthread method

3.2.4 Compare MPI implementation and Pthread implementation

In this part, I will compare MPI implementation and Pthred method. From figures showing above, we can learn that the execution time of MPI is less than Pthread when the input size is very small and the execution time of MPI is larger than Pthread when input size is large.

First, theoretically analyze the execution of Pthread implementation and MPI im-

plementation. It is obvious that the execution time of Pthread should be smaller because Pthread implementation do not need to spend time on inter thread communication. All threads share the same memory space, and they just need to compute the final result and put the result to the final place during calculation process.

However, because MPI implementation cannot share memory, master process should send data to each process and let them do computation. This inter process communication is time consuming. At the same time, according to my implementation, in order to scatter data to each process, master process also process input data before scattering data to all process. This process also consumes some time.

Second, why when input data is small, the MPI has smaller execution time? One possible reason is that compiler do optimization by itself which means the execution time is faster than theoretical analysis.

3.2.5 Potential optimization method

After executing GUI version of program, we can find that different part of data have different workload. In order to improve the performance of the program, each process or thread should have similar workload, which means they'd better separate those black part of data equally. Nevertheless, how to achieve this?

For MPI, we can separate the whole big input data into many pieces of small data block. When sending data to each process, the program can those parts sequentially. Compared with separate the whole data input to process number blocks, break the whole input data array to many small piece like each part only has 3 data point can make sure each process can get some point for those hardest part.

For Pthread, we can also break the whole process to lots of pieces of data block. Each thread can get data from main memory when they finish previous computation. This can make sure all thread are working rather than some threads finish their work and wait other threads. According to this implementation, all threads finish at similar time, which can improve the whole program performance by adding a mutex lock.

4 Conclusion

In this project, I learn the basic parallel computing model and utilizing MPI and Pthread to implement a parallel execution code. At the same time, I also tried to debug a parallel execution code with different implementation methods. During implementing the program, I learn how processes communicate with each other, how one process send and receive data from other processes and how threads created by one process share memory.

After finishing the test experiment and analyzing result, I found that more process number or more thread number used in parallel computing does not means better performance. At the same time, I also find the difference between performance of MPI and Pthread implementation. We can see that when the communication between processes increase, the execution time would also increase.

Finally, after calculating the efficiency, I found that parallel computing can use less execution time, but it need more resource and the efficiency shows that parallel computing is less efficient than sequential execution according my implementation. And the more the inter process or inter thread communication are, the less efficiency is.

5 Appendix

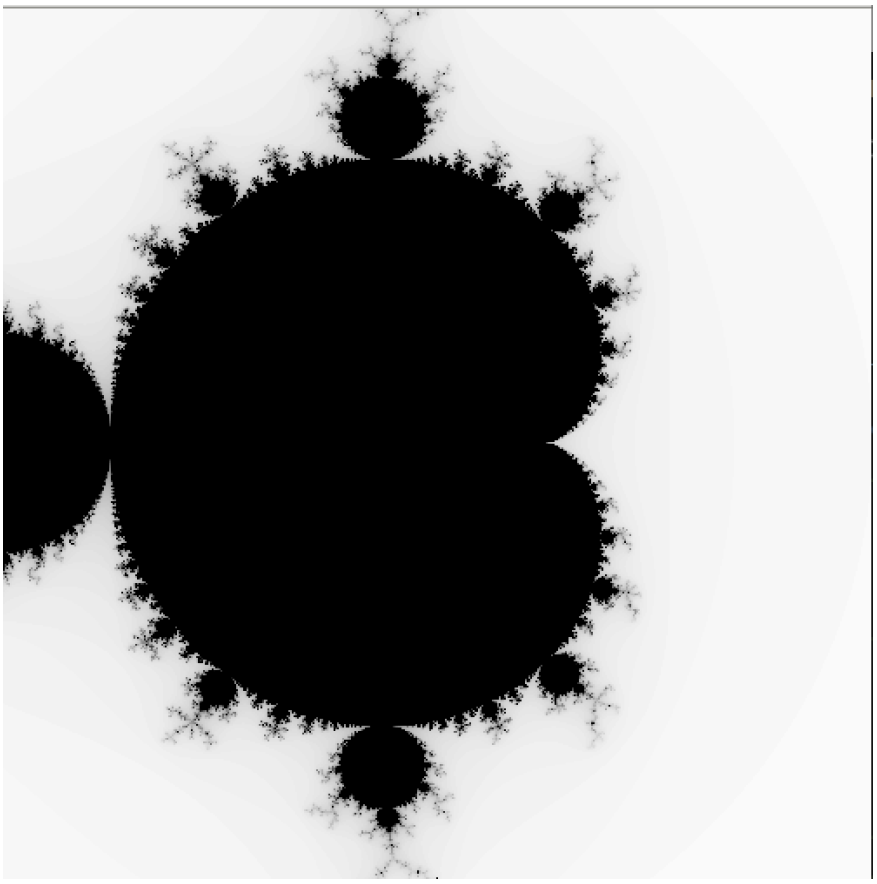


Figure 15: GUI output of input size with 800 * 800 with maximum 100 loop number

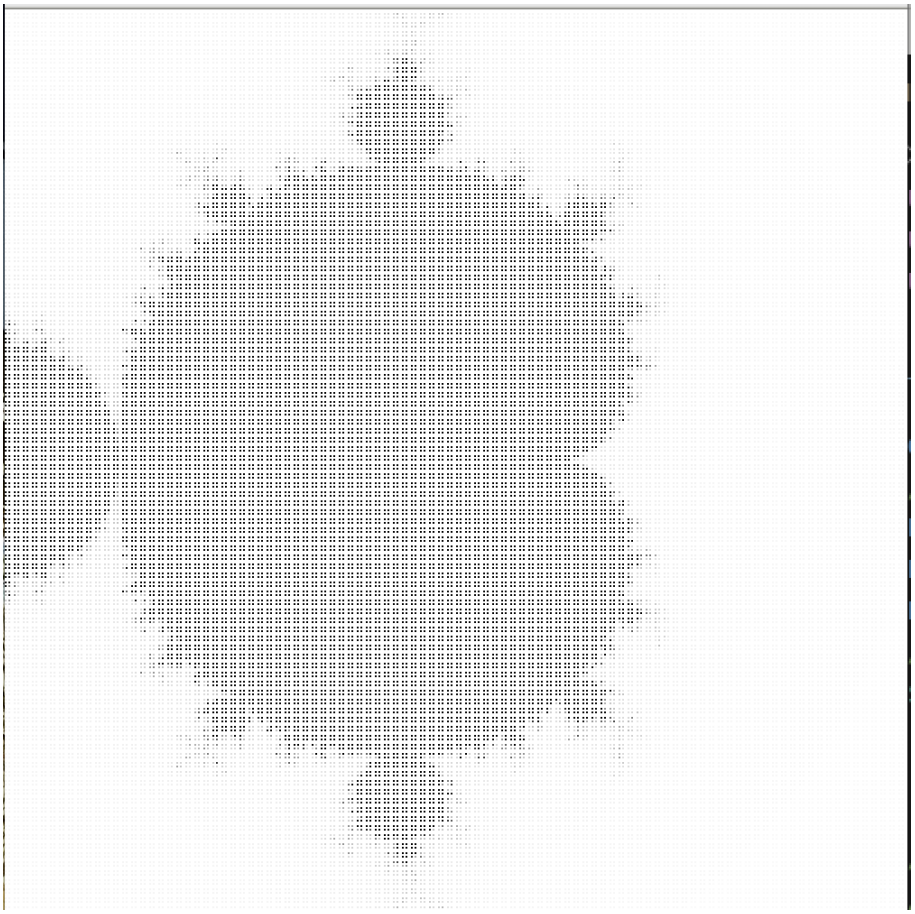


Figure 16: GUI output of input size with 200 * 200 with maximum 100 loop number

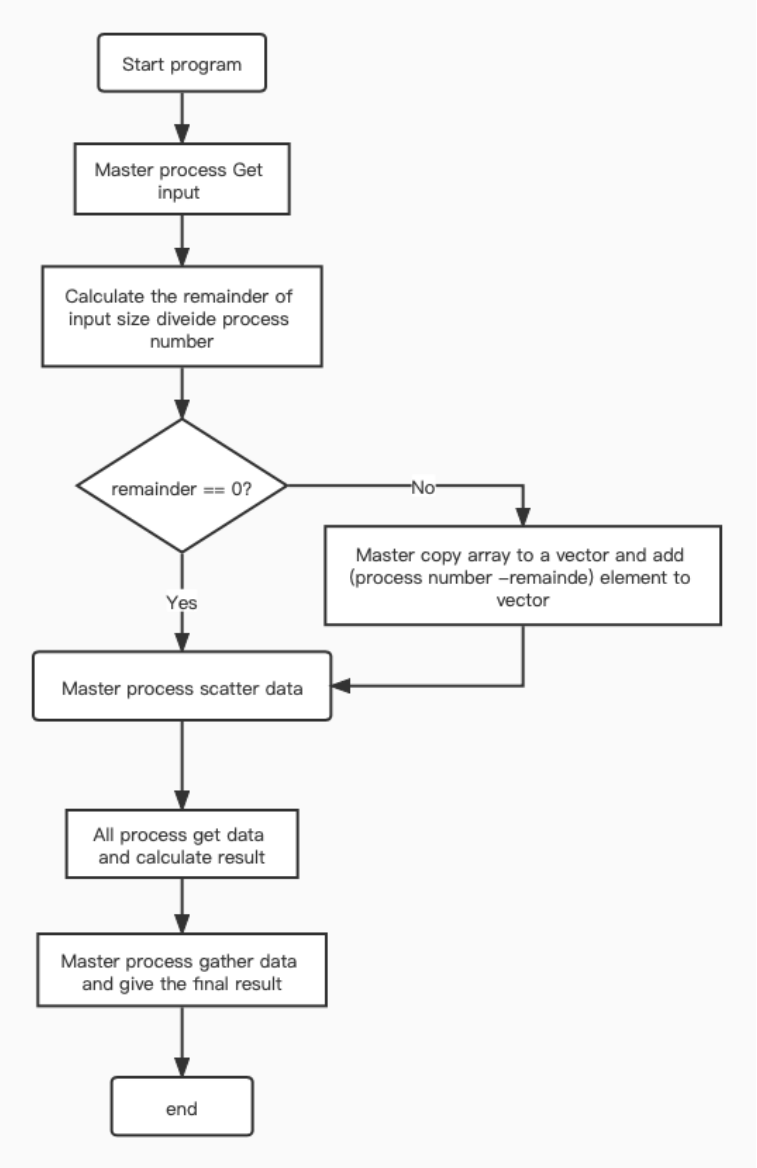


Figure 17: Algorithm flow chart of MPI implementation

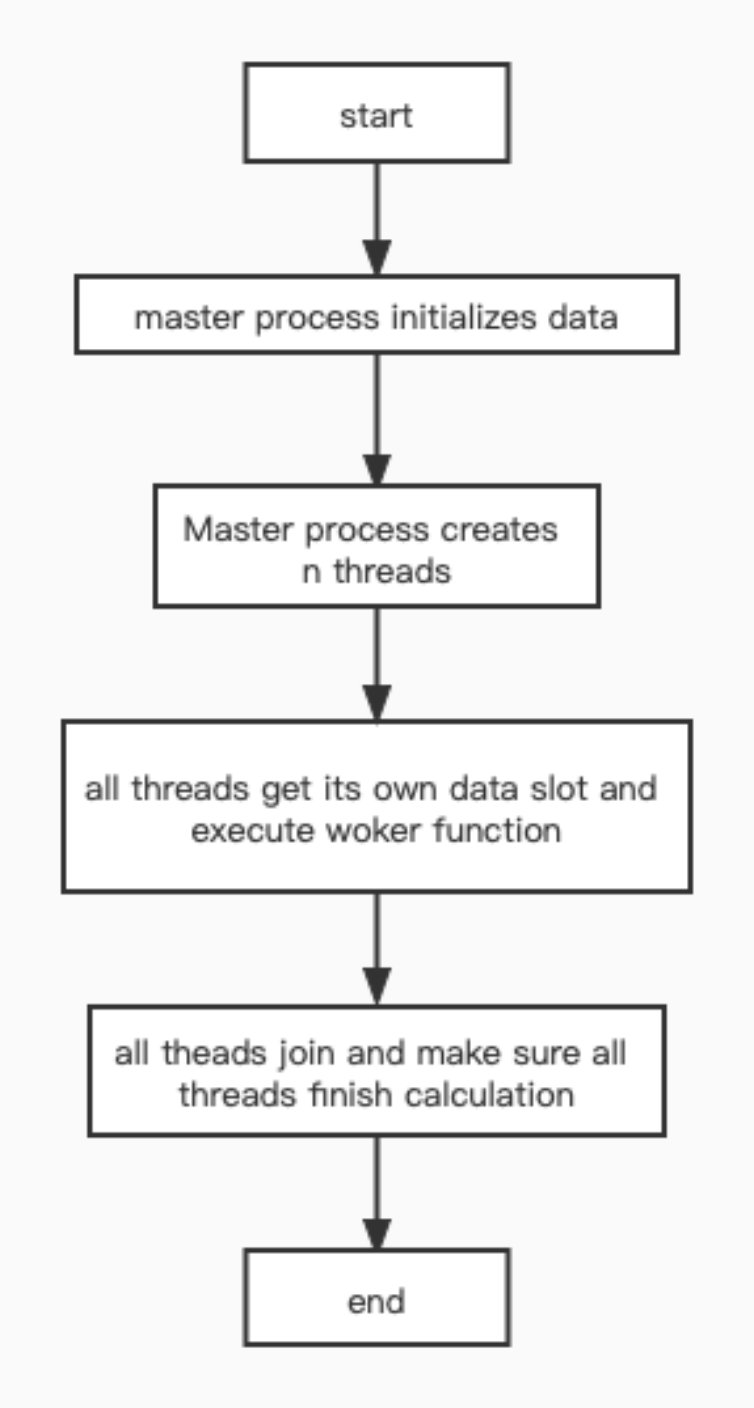


Figure 18: Algorithm flow chart of Pthread implementation