The Chinese University of Hong Kong, Shenzhen

CSC4005

Parallel and Distributed Computing

Project4
Heat Distribution

Author:
Zijuan Lin

Student Number:
121090327

December 5, 2022

Contents

1 Running the program

In this section, we will introduce how to run the programs provided.

Sequential Implementation To run the Sequential Implementation, you shall compile the source code using **make seq** command to get the program without GUI, or **make seqg** to get the one with GUI. Then use
./seq {# of problem size}
to run the program without GUI or
./seqg {# of problem size}
to run the program with GUI.

Result:

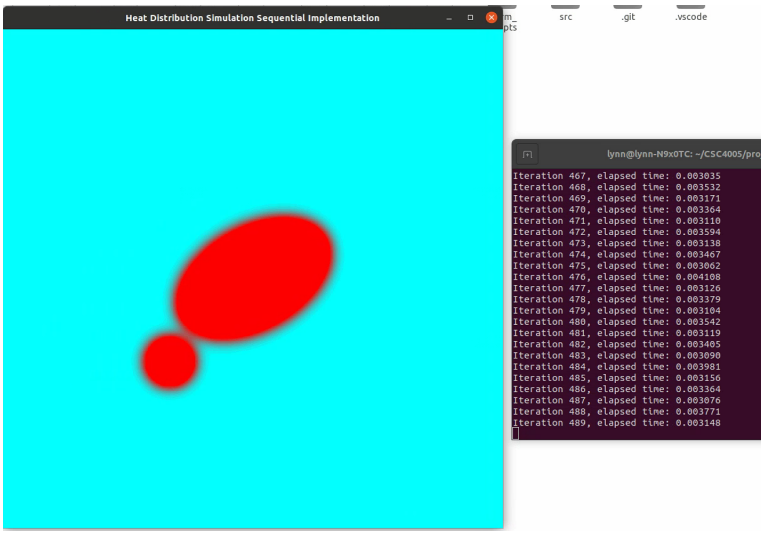

Figure 1: GUI of Sequential Implementation

MPI Implementation To run the MPI Implementation, you shall compile the source code using **make mpi** command to get the program without GUI, or **make mpig** to get the one with GUI. Then use
mpirun -np {# of proc} ./mpi {# of problem size}
to run the program without GUI or
mpirun -np {# of proc} ./mpig {# of problem size}
to run the program with GUI.

Result:

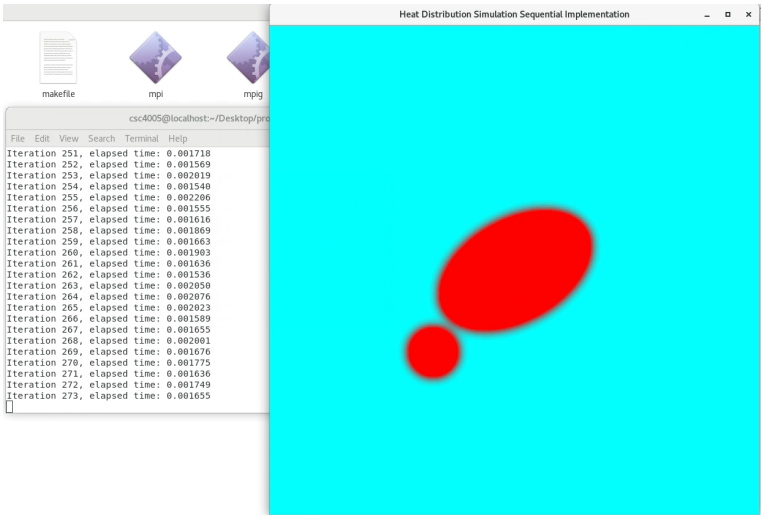

Figure 2: GUI of MPI Implementation

Pthread Implementation To run the Pthread Implementation, you shall compile the source code using **make pthread** command to get the program without GUI, or **make pthreadg** to get the one with GUI. Then use
./pthread {# of problem size} {# of threads}
to run the program without GUI or
./pthreadg {# of problem size} {# of threads}
to run the program with GUI.

Result:

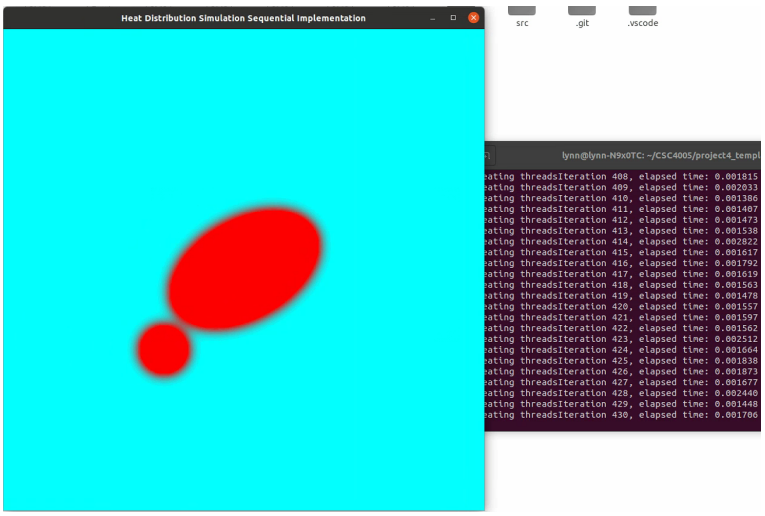

Figure 3: GUI of Pthread Implementation

OpenMP Implementation To run the OpenMP Implementation, you shall compile the source code using **make openmp** command to get the program without GUI, or **make openmpg** to get the one with GUI. Then use
./openmp {# of problem size} {# of threads}
to run the program without GUI or
./openmpg {# of problem size} {# of threads}
to run the program with GUI.

Result:

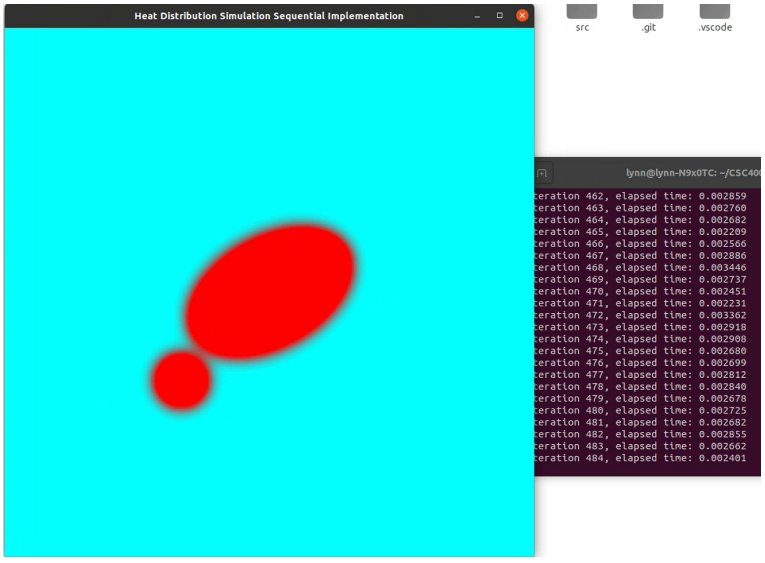

Figure 4: GUI of OpenMP Implementation

CUDA Implementation To run the CUDA Implementation, you shall compile the source code using **make cuda** command to get the program without GUI, or **make cudag** to get the one with GUI. Then use
./cuda {# of problem size}
to run the program without GUI or
./cudag {# of problem size}
to run the program with GUI.

Result:

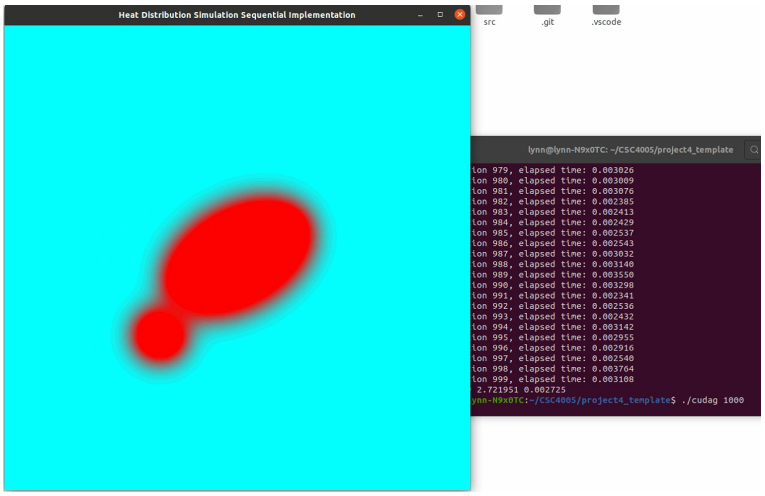

Figure 5: GUI of CUDA Implementation

OpenMP+MPI Implementation To run the OpenMP+bonus Implementation, you shall compile the source code using **make bonus** command to get the program without GUI, or **make bonusg** to get the one with GUI. Then use
mpirun -np {# of proc} ./bonus {# of problem size} {# of threads}
to run the program without GUI or
mpirun -np {# of proc} ./bonusg {# of problem size} {# of threads}
to run the program with GUI.

Result:

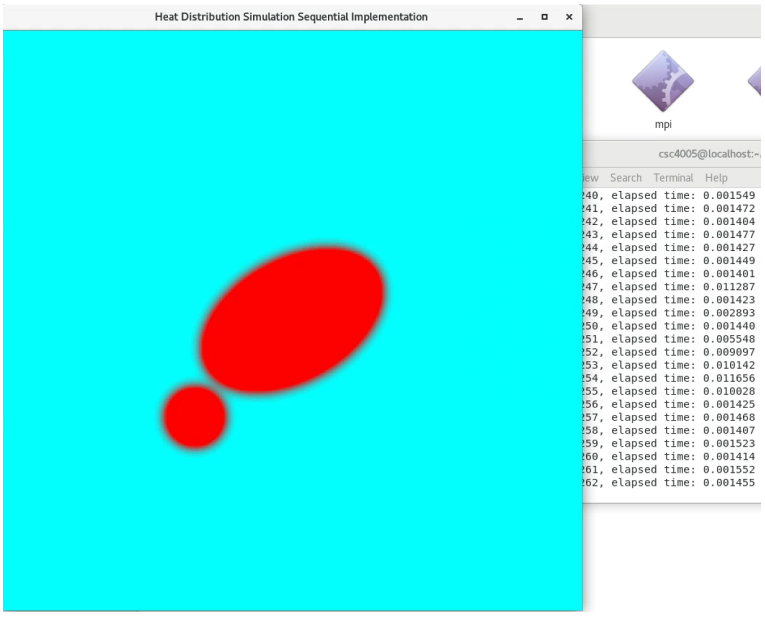

Figure 6: GUI of MPI+OpenMP Implementation

2 Program Structure

Problem Statement In the room there are some fireplaces releasing heat to the surrounding area. The heat distribution process can be described by **Jacobi Iteration**, which is the following formula:

$$h_{(i,j)} = \frac{h_{(i-1,j)} + h_{(i+1,j)} + h_{(i,j-1)} + h_{(i,j+1)}}{4}$$

mpig Here $h_{(i,j)}$ denotes the temperature at point (i, j). This is an algorithm with time complexity of $O(n^2)$, where n denotes the number of points.

In the problem, we are required to write programs to do the iteration process and show the temperature change in GUI.

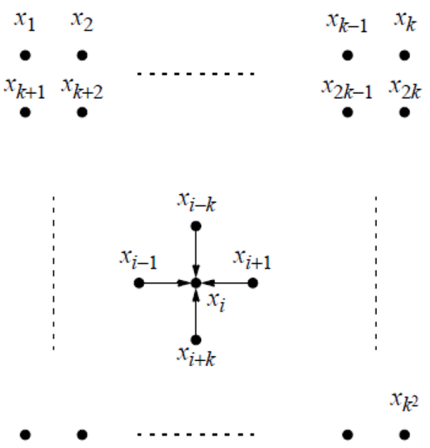

Figure 7: Natural Ordering of Heat Distribution Problem

MPI Implementation In the MPI program, the structure can be described by the following graph.

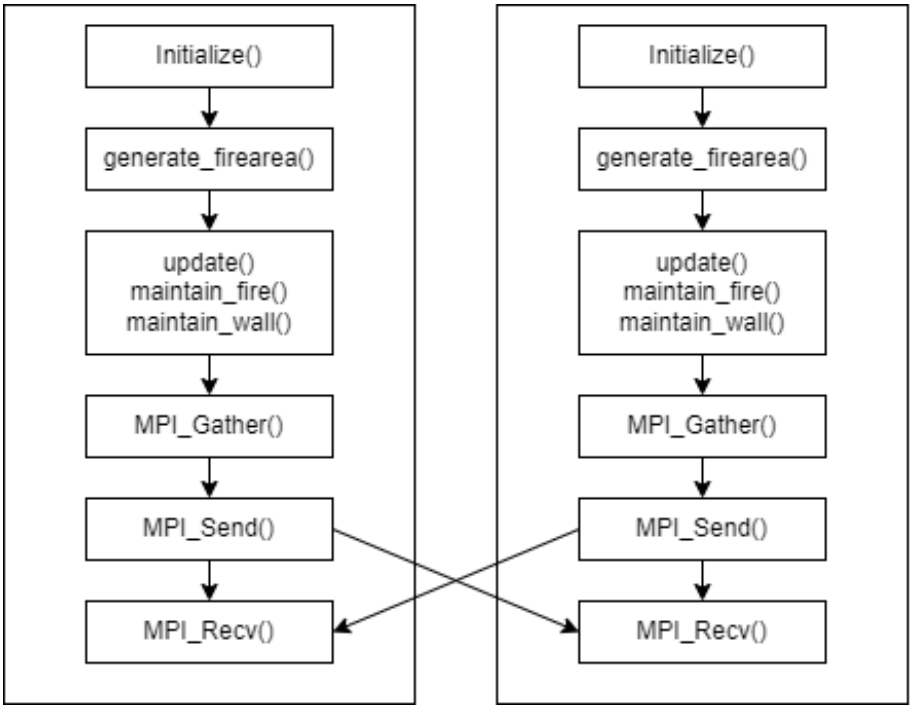

Figure 8: The structure of MPI Implementation

In each process, they **Initialize()** and **Generate_firearea()** individually, and update the points with different begin position:

int my_size=size*size/world_size+1;
int start=my_size*my_rank;
int end=(my_rank+1)*my_size >size*size ? size*size : (my_rank+1)*my_size;

Then the results are gathered in Rank 0 order to render graphs. Ranks will also use **MPI_Send()** and **MPI_Recv()** to send and receive their first and last row they are in charge of to rank-1 and rank+1 respectively.

Pthread Implementation The structure of Pthread Implementation can be described by the following graph.

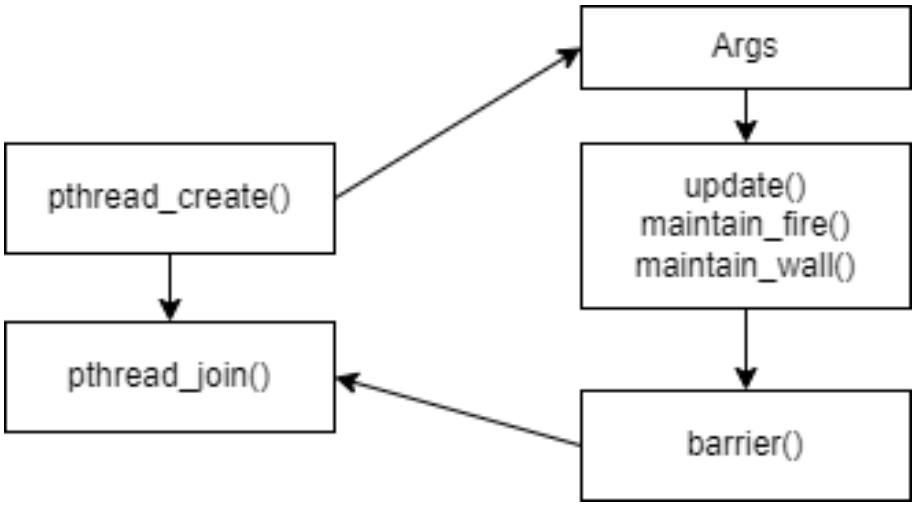

Figure 9: The structure of Pthread Implementation

The **Args** each thread receives is a struct consists of the following elements.

typedef struct { int my_thd; //describes the # of the thread
int n_thd; //describes the total number of threads
int size; //problem size
int count; //current iteration count
float *odd; //pointer to the array storing data
float *even; //pointer to the array storing results
bool *fire_area;
} Args;

OpenMP Implementation In the OpenMP Implementation, tasks are distributed in the following approach.

```
omp_set_num_threads(n_omp_threads);
#pragma omp parallel for
for (int i=0;i<size*size-1;i++){}
```

CUDA Implementation The structure of CUDA Implementation can be described by the following graph.

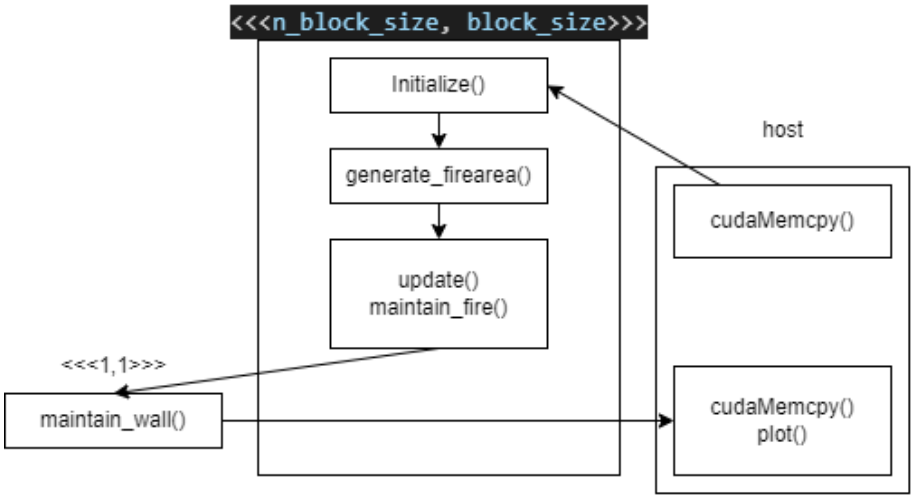

Figure 10: The structure of CUDA Implementation

3 Performance Analysis

The results are tested on HPC with **Intel(R) Xeon(R) Silver 4210R CPU @ 2.40GHz**, CUDA implementation is tested with **GTX 1660ti** with **CUDA 10.2**.

Overall comparison The overall performance is as follows. Every implementation runs 1000 iterations. In the first figure, execution time of MPI Implementation using more than 20 processes is not included due to overspawn.

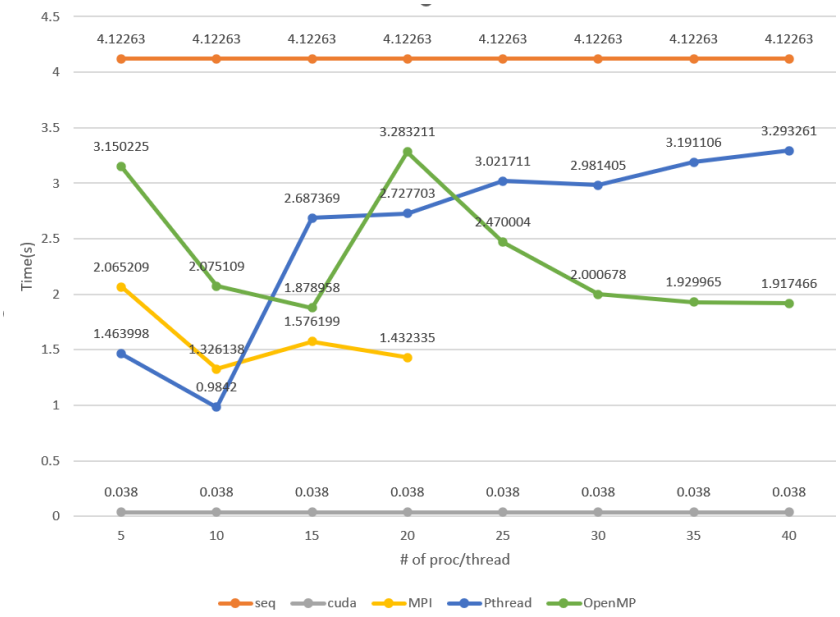

Figure 11: The overall performance without overspawn

Due to overspawn, the execution time of MPI Implementation is significantly affected.

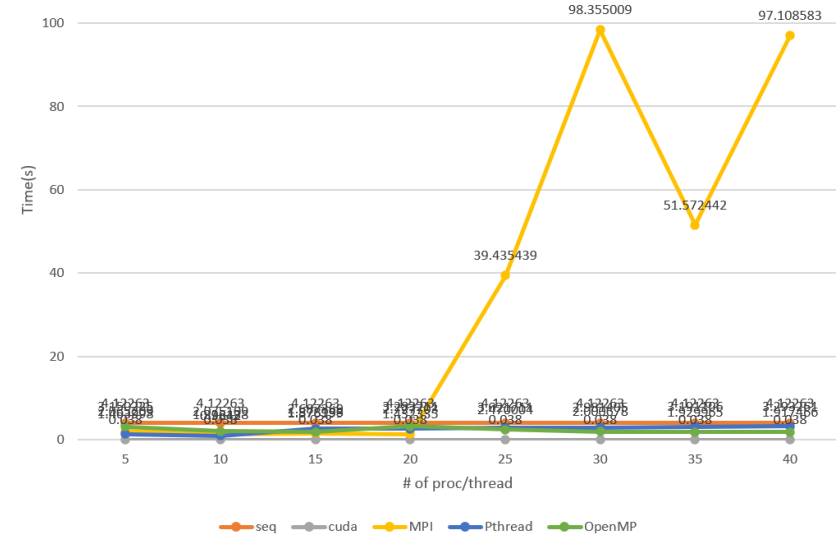

Figure 12: The overall performance with overspawn

Pthread & OpenMP However, since Ubuntu can create at most 32767 threads, we can generate more threads to test the implementations.

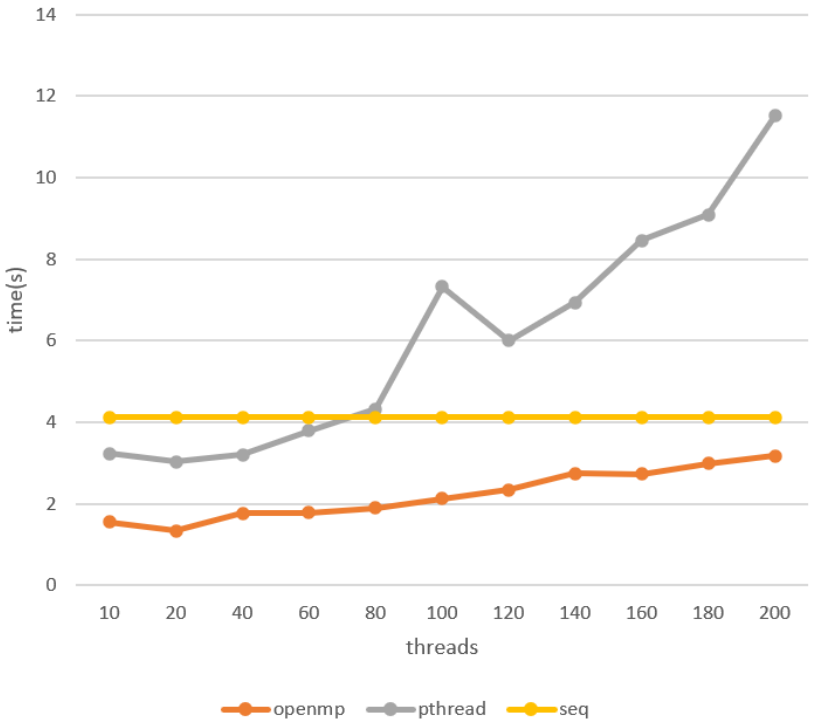

Figure 13: The performance of Pthread and OpenMP

From the above figure we can see that the time cost of inter-thread communication as well as creating and joing threads can also significantly affect the performance.

Problem size & Execution time From the analysis above, we can see that the time complexity of the algorithm is $O(n^2)$. Here we take the MPI Implementation as an example to illustrate the influence of increasing problem size. The following figure shows the execution time of MPI implementation in different problem size using 20 processes,and the iteration number is still 1000.

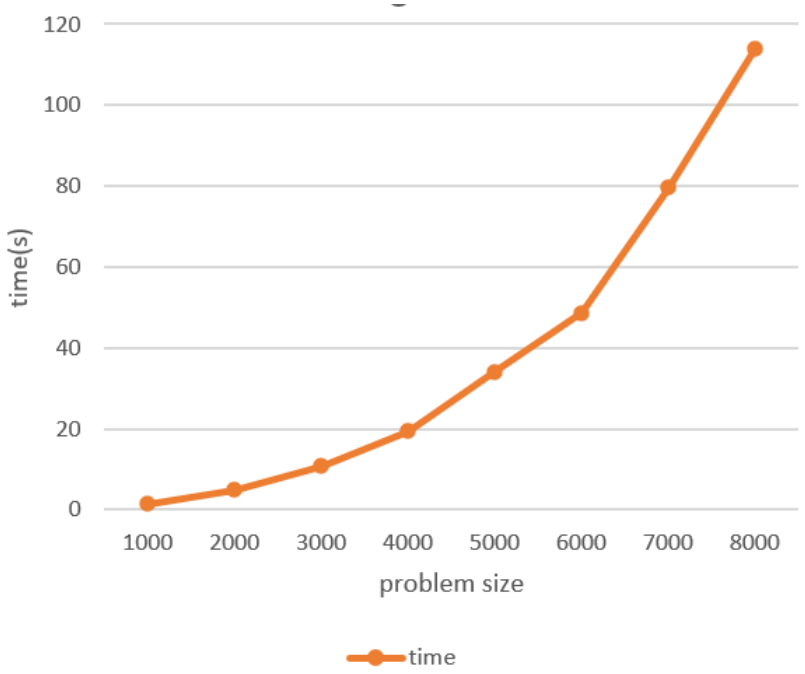

Figure 14: The performance MPI Implementation in different problem size

We can see that the execution time is growing quadraticly, which proves the analysis of time complexity.

4 Conclusion

The Heat Distribution calculation is more difficult than N-Body Simulation since the Read and Write processes are both done on the value (temperature) of points. Much efforts have to be devoted to design a slightly complicated structure in order to avoid data race. However, the inter-process communication will be less time-consuming if no GUI rendering is needed because each process only needs to obtain two rows from the previous and latter process.