

# CSC4005 Assignment4 Report

## Introduction

The objective of this assignment is to simulate heat diffusion in a room. Initially, there is a fireplace with  $100^{\circ}C$  and all other places including four walls are set to be  $20^{\circ}C$ . The temperature of the initial fireplace and the walls remains unchanged during the whole process. The temperature inside the room is going to be calculated by using Jacobi iteration and it would be plotted in color. The iteration algorithm is described as follows:

$$(h_t)_{i,j} = \frac{1}{4} \times ((h_{t-1})_{i-1,j} + (h_{t-1})_{i+1,j} + (h_{t-1})_{i,j-1} + (h_{t-1})_{i,j+1}) \quad (1)$$

where  $h_{t-1}$  represents the temperature in the last iteration and  $(i, j)$  indicates the coordinate of the points.

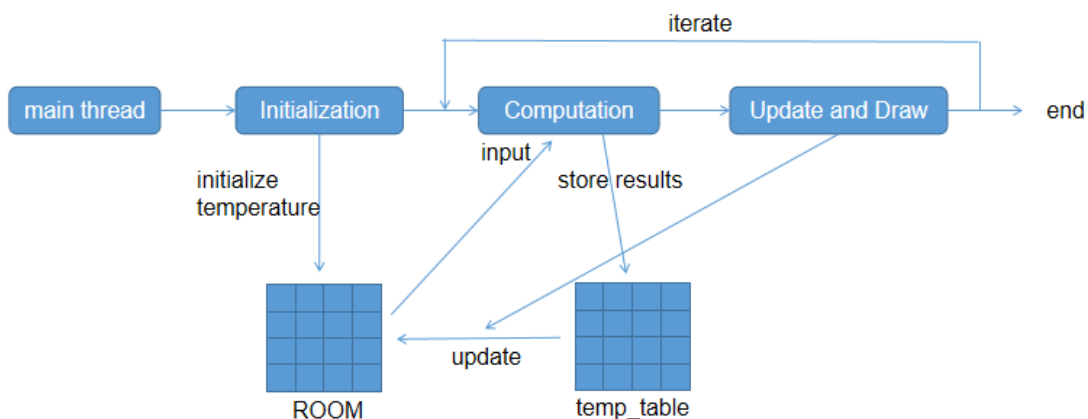
In this experiment, five versions of programs have been designed: sequential implementation, Pthread implementation, MPI implementation, OpenMP implementation and MPI+OpenMP version. The source code has been attached to this report.

This report is going to elaborate the detailed implementations and analyze the performance of different implementations.

## Program design

The logic of the sequential version is the basis of the logic of parallel versions. Generally, the parallel versions derive from the sequential version by parallelize the parallelizable work and keep the parts that are not parallelizable. Thus, in this part, the detailed implementation of heat simulation in sequential version would be elaborated, but for parallel versions, we only focus on how to parallelize the work.

### Sequential version



The graph above shows the basic logic of the heat simulation with sequential version. There are several main work to be done in order, the detail is as follows:

1. Initialization of the temperature inside the room and prepare a table to store the calculated results temporarily:

```

// initialize the temperature inside the room
double *ROOM = (double *) malloc(sizeof(double)*(X_RESN*Y_RESN));
for (int i = 0; i < X_RESN; ++i){
    for (int j = 0; j < Y_RESN; ++j){
        // initialize temperature of room and walls
        ROOM[i*X_RESN + j] = 20;
        // initialize temperature of fire place
        if (i == 0 && j > X_RESN*2/5 && j < X_RESN*3/5)
            ROOM[i*X_RESN + j] = 100;
    }
}
// prepare a table to temporarily store the results
double *temp_table = (double *) malloc(sizeof(double)*(X_RESN*Y_RESN));

```

2. Compute the temperature of each point in the room and store the result into the temp\_table, since directly updating the ROOM table would affect the results of the points that have not computed yet in the same round of iteration:

```

// computing the temperature
for (int i = 0; i < X_RESN; ++i){
    for (int j = 0; j < Y_RESN; ++j){
        if (i == 0 || j == 0 || i == SIZE-1 || j == SIZE-1){ // walls
            temp_table[i*SIZE + j] = ROOM[i*SIZE + j];
        }else{
            temp_table[i*SIZE + j] = 0.25 * (ROOM[(i-1)*SIZE + j] +
ROOM[(i+1)*SIZE + j] + ROOM[i*SIZE + j-1] + ROOM[i*SIZE + j+1]); // store
in temp_table
        }
    }
}

```

3. Update the ROOM table based on temp\_table and plot the results:

```

double temperature;
long unsigned c_pixel;
for (int i = 0; i < X_RESN; ++i){
    for (int j = 0; j < Y_RESN; ++j){

        ROOM[i*X_RESN + j] = temp_table[i*X_RESN + j]; // update ROOM

        if (count % 10 != 0) continue; // plot every 10 iterations

        temperature = ROOM[i*X_RESN + j];
        c_pixel = get_cpixel(temperature); // get color.pixel based on
temperature
        XSetForeground (display, gc, c_pixel);
        XDrawPoint (display, win, gc, i, j);
    }
}
if (count % 10 != 0) continue; // plot every 10 iterations
XFlush (display);

```

Here, the function *get\_cpixel(temperature)* is defined as follows:

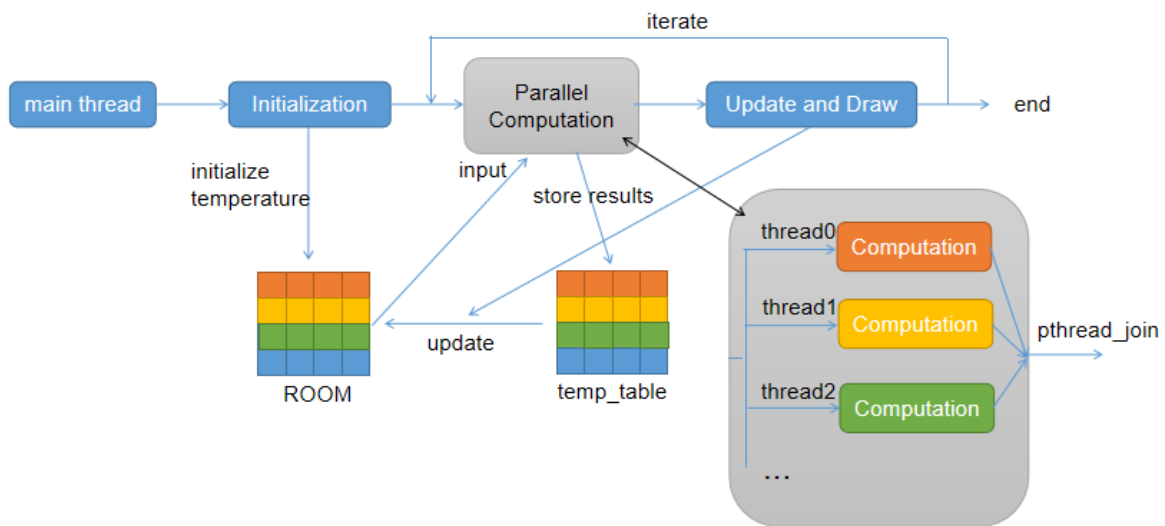
```
// preset color.pixel based on temperature
unsigned long get_cpixel(double temperature){
    unsigned long c_pixel = 3014865;
    int temp = (int)temperature;
    int r = (temp - 20) / 5;
    c_pixel += 786420 * r;
    return c_pixel;
}
```

The parameters in this definition were adjusted from function *XAllocColor()*, which could display different color at  $5^{\circ}C$  intervals with red and blue. Since function *XAllocColor()* runs slow, the parameters were recorded and used for creating this *get\_cpixel()* function.

4. Repeat step 2 and 3 for a certain times (iteration):

```
int iter = 0;
while (iter++ < ITERATIONS){
    // [code of step 2 and 3]
}
```

### Pthread version



The graph above shows the logic of pthread implementation. Comparing with the sequential version, the only change is that the computation part was parallelized by using multiple threads. All the threads are going to operate on the same tables but with different target rows. The color in the graph shows the target rows for each threads.

In this parallelization, two main works were done:

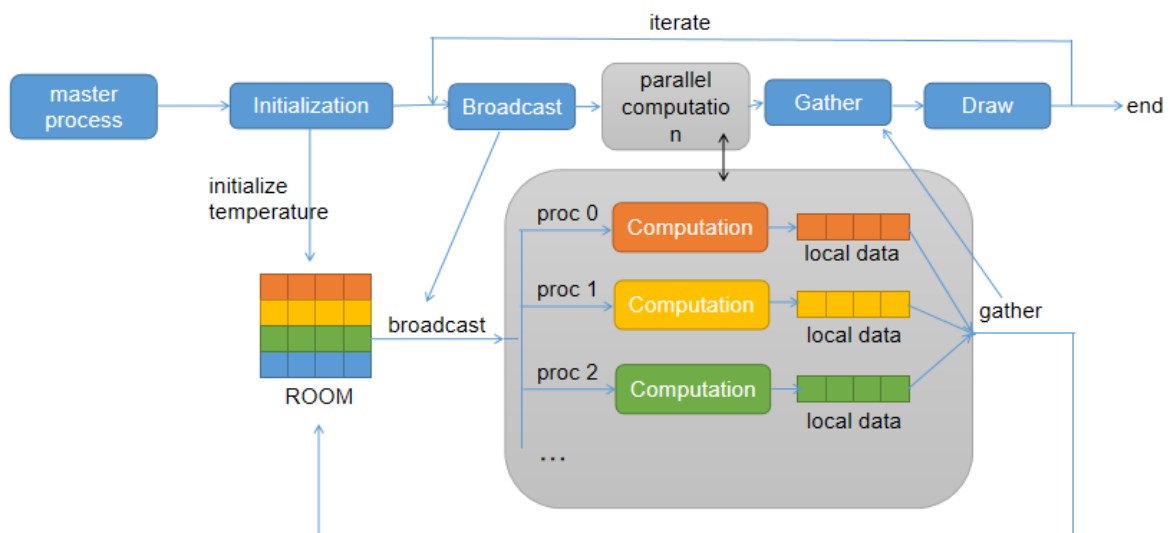
- Based on thread id, each thread specifies its target rows for computation:

```
// specify the target rows
num = SIZE / NUM_THREADS;
remain = SIZE % NUM_THREADS;
if (tid < remain){
    num ++;
    start = num * tid;
}else{
    start = num * tid + remain;
}
end = start + num;
```

- The main thread distribute the tasks and collect the threads after computation:

```
// distribute the computation work
for (i = 0; i < NUM_THREADS; ++ i){
    rc = pthread_create(&thread[i], NULL, cal_func, &input_data[i]);
    if (rc) {
        fprintf(stderr, "error: pthread_create, rc: %d\n", rc);
        return EXIT_FAILURE;
    }
}
// gather the threads
for (i = 0; i < NUM_THREADS; ++i) {
    pthread_join(thread[i], &return_data);
    // printf("the return data is %d\n", return_data);
}
```

## MPI version



The graph above shows the logic of MPI implementation. Different with the sequential version, after the initialization, the master process broadcasts the whole ROOM table to all processes and all processes do the computation on their target rows. The results computed in each process are stored in the local memory and then gathered to the master process to update the ROOM table.

In this parallelization, three main works were done:

- Broadcast the data to all process:

```
// Broadcast the updated ROOM to every node
MPI_Bcast(ROOM, SIZE*SIZE, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

- Specify the target rows for computation and specify the displacement list for later gathering:

```
disp[0] = 0;
int quotient = SIZE / numtasks;
int remainder = SIZE % numtasks;

for (int i = 0; i < numtasks; ++ i) {
    if (i < remainder) {
        count[i] = (quotient + 1) * SIZE;
    }else{
        count[i] = quotient * SIZE;
    }
    if (i > 0) disp[i] = disp[i-1] + count[i-1];
}

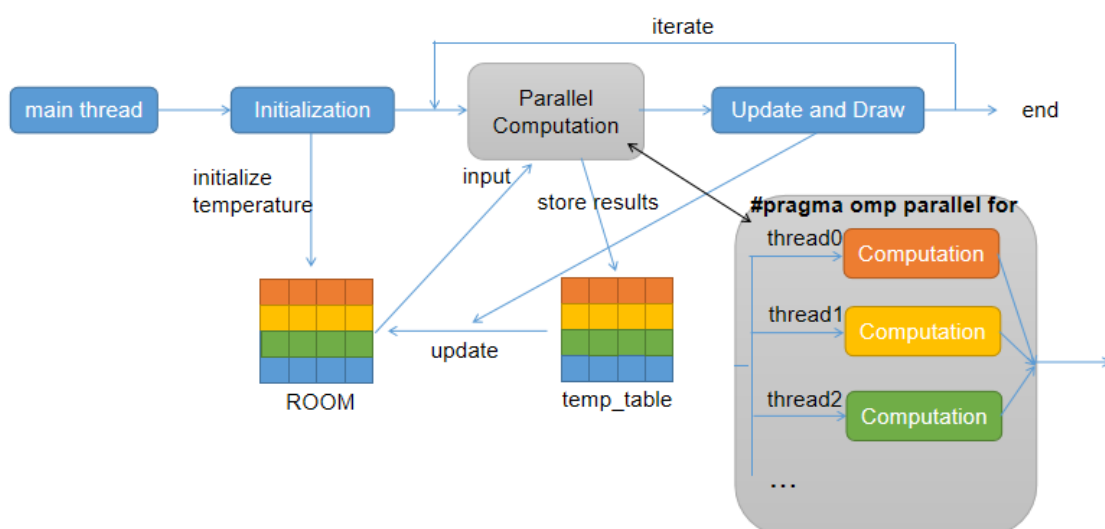
// the array that stores the result of computation
local_data = (double *)malloc(sizeof(double) * count[taskid]);

// the index of the first data of the first target body
int start_row = disp[taskid] / SIZE;
// the index of the first data of the body after the last target body
int end_row = start_row + count[taskid] / SIZE;
```

- Gather the results from each process:

```
MPI_Gatherv(local_data, count[taskid], MPI_DOUBLE, ROOM, count, disp,
MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

## OpenMP version



For OpenMP version, its logic is almost the same as the pthread version, but it has simpler implementation.

To parallelize the work, simply add a line of code ahead of the for loop:

```
#pragma omp parallel for private(j) shared(temp_table, ROOM)
```

## MPI+OpenMP version

Based on the MPI version, the for loop of initialization in the master process and the for loop of computation in all processes were parallelized by OpenMP.

## How to run

### Sequential version

1. In the directory of source code, type the following command and enter to compile the program:

```
g++ heat_simulation_seq.cpp -lX11 -o ./hw4_seq.out
```

then an executable file *hw4\_seq.out* will be generated in the directory.

2. In the same directory, type the following command and enter to run the program:

```
./hw4_seq.out (SIZE)
```

where SIZE is a optional parameter that could specify the size of the windows as  $\text{SIZE} \times \text{SIZE}$ .

Default setting: SIZE = 200.

### Pthread version

1. In the directory of source code, type the following command and enter to compile the program:

```
g++ heat_simulation_pthread.cpp -lX11 -lpthread -o ./hw4_pthread.out
```

then an executable file *hw4\_pthread.out* will be generated in the directory.

2. In the same directory, type the following command and enter to run the program:

```
./hw4_pthread.out (SIZE) (NUM_THREADS)
```

where SIZE and NUM\_THREADS are optional parameters that could specify the size of the window and the number of threads used for simulation.

(NOTE: to specify NUM\_THREADS, SIZE should be specified first)

Default setting: SIZE= 200, NUM\_THREADS = 2.

### MPI version

1. In the directory of source code, type the following command and enter to compile the program:

```
mpic++ heat_simulation_mpi.cpp -lX11 -o ./hw4_mpi.out
```

then an executable file *hw4\_mpi.out* will be generated in the directory.

2. In the same directory, type the following command and enter to run the program:

```
mpirun -np N ./hw4_mpi.out (SIZE)
```

where N specifies the number of processes and SIZE is optional to specify the size of the window as  $\text{SIZE} \times \text{SIZE}$ .

Default setting: NUM\_BODIES = 200.

### OpenMP version

1. In the directory of source code, type the following command and enter to compile the program:

```
g++ heat_simulation_openmp.cpp -fopenmp -lX11 -o ./hw4_openmp.out
```

then an executable file *hw4\_openmp.out* will be generated in the directory.

2. In the same directory, type the following command and enter to run the program:

```
./hw4_openmp.out (SIZE) (NUM_THREADS)
```

where SIZE and NUM\_THREADS are optional parameters that could specify the size of the window and the number of threads used for simulation.

(NOTE: to specify NUM\_THREADS, SIZE should be specified first)

Default setting: SIZE= 200, NUM\_THREADS = 2.

### MPI+OpenMP version

1. In the directory of source code, type the following command and enter to compile the program:

```
mpic++ heat_simulation_mpi+openmp.cpp -fopenmp -lX11 -o ./hw4_bonus.out
```

then an executable file *hw4\_bonus.out* will be generated in the directory.

2. In the same directory, type the following command and enter to run the program:

```
mpirun -np N ./hw4_mpi.out (SIZE) (NUM_THREADS)
```

where N specifies the number of processes , SIZE and NUM\_THREADS are optional to specify the size of the window and the number of threads used for simulation.

(NOTE: to specify NUM\_THREADS, SIZE should be specify first)

Default setting: SIZE = 200, NUM\_THREADS = 2.

## Sample output

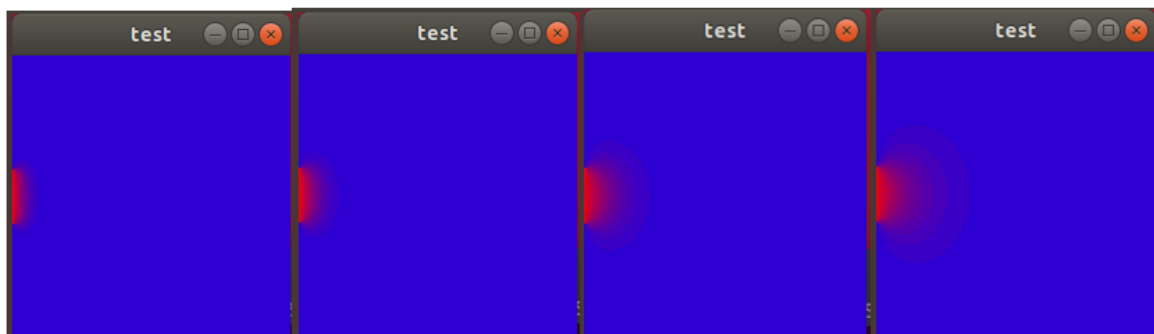
### Running environment

The sample is running on the virtual machine (with *Xlib* and *mpich-3.2.1* installed):



## Sample display

The heat would diffuse from the center of the left wall. Red represents high temperature and blue represents low temperature. The color contours at  $5^\circ C$  intervals. It would display as the following screen shots. Also, a sample video has been attached with this report.



## Sequential version

```
cuhksz_csc@CSC4005: ~/Documents
File Edit View Search Terminal Help
cuhksz_csc@CSC4005:~/Documents$ g++ heat_simulation_seq.cpp -lX11 -o ./hw4_seq.out
heat_simulation_seq.cpp: In function 'int main(int, char**)':
heat_simulation_seq.cpp:28:36: warning: ISO C++ forbids converting a
rite-strings]
    char *window_name = "test", *display_name = NULL;
ion for a window */

cuhksz_csc@CSC4005:~/Documents$ ./hw4_seq.out
Name: Hongpeng Yi
Student ID: 117020343
Assignment 4, Heat Simulation, Sequential implementation.
RunTime is 7.992815s.
Window size is 200*200; total 20000 iterations.
█
```

## Pthread version

```
cuhksz_csc@CSC4005: ~/Documents
File Edit View Search Terminal Help
cuhksz_csc@CSC4005:~/Documents$ g++ heat_simulation_pthread.cpp -lX11 -lpthread -o ./hw4_pthread.out
heat_simulation_pthread.cpp: In function 'int main(int, char**)':
heat_simulation_pthread.cpp:75:36: warning: ISO C++ forbids conver
[-Wwrite-strings]
    char *window_name = "test", *display_name = NULL;
ion for a window */

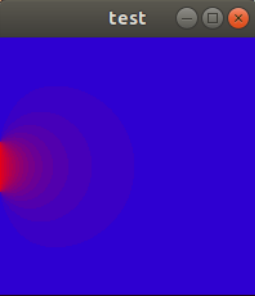
cuhksz_csc@CSC4005:~/Documents$ ./hw4_pthread.out 200 2
Name: Hongpeng Yi
Student ID: 117020343
Assignment 4, Heat Simulation, Pthread implementation.
RunTime is 18.043116s.
Window size is 200*200; total 20000 iterations.
█
```

## MPI version



```
cuhksz_csc@CSC4005: ~/Documents
File Edit View Search Terminal Help
cuhksz_csc@CSC4005:~/Documents$ mpic++ heat_simulation_mpi.cpp -lX11 -o ./hw4_mpi.out
heat_simulation_mpi.cpp: In function 'int main(int, char**)':
heat_simulation_mpi.cpp:29:36: warning: ISO C++ forbids converting a string literal to 'char*' [-Wwrite-strings]
    char *window_name = "test", *display_name = NULL;
                                ^
/* initialization for a window */

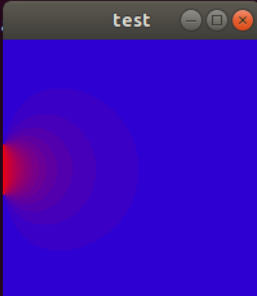
cuhksz_csc@CSC4005:~/Documents$ mpirun -np 2 ./hw4_mpi.out
Name: Hongpeng Yi
Student ID: 117020343
Assignment 4, Heat Simulation, MPI implementation.
RunTime is 4.871345s, with 2 processes.
Window size is 200*200; total 20000 iterations.
```



## OpenMP version

```
cuhksz_csc@CSC4005: ~/Documents
File Edit View Search Terminal Help
cuhksz_csc@CSC4005:~/Documents$ g++ heat_simulation_openmp.cpp -lX11 -fopenmp -o ./hw4_omp.out
heat_simulation_openmp.cpp: In function 'int main(int, char**)':
heat_simulation_openmp.cpp:33:36: warning: ISO C++ forbids converting a string literal to 'char*' [-Wwrite-strings]
    char *window_name = "test", *display_name = NULL;
                                ^
/* initialization for a window */


cuhksz_csc@CSC4005:~/Documents$ ./hw4_omp.out 200 2
Name: Hongpeng Yi
Student ID: 117020343
Assignment 4, Heat Simulation, OpenMP implementation.
RunTime is 5.722396s, with 2 threads.
Window size is 200*200; total 20000 iterations.
```



## MPI+OpenMP version

```
cuhksz_csc@CSC4005: ~/Documents
File Edit View Search Terminal Help
cuhksz_csc@CSC4005:~/Documents$ mpic++ heat_simulation_mpi+openmp.cpp -lX11 -fopenmp -o ./hw4_bonus.out
heat_simulation_mpi+openmp.cpp: In function 'int main(int, char**)':
heat_simulation_mpi+openmp.cpp:31:36: warning: ISO C++ forbids converting a string literal to 'char*' [-Wwrite-strings]
    char *window_name = "test", *display_name = NULL;
                                ^
/* initialization for a window */

cuhksz_csc@CSC4005:~/Documents$ mpirun -np 2 ./hw4_bonus.out 200 2
Name: Hongpeng Yi
Student ID: 117020343
Assignment 4, Heat Simulation, MPI+OpenMP implementation.
RunTime is 174.239523s, with 2 processes and 2 threads.
Window size is 200*200; total 20000 iterations.
```



## Performance analysis

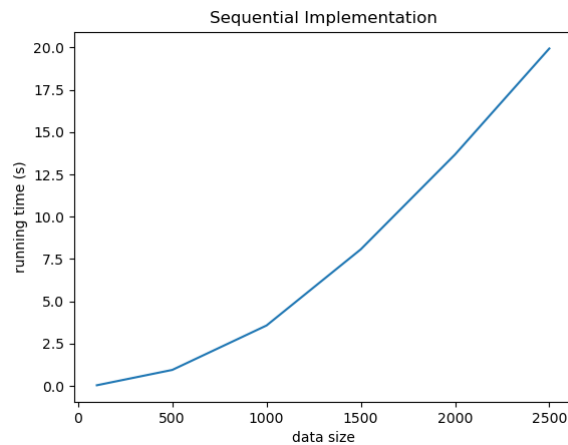
In this part, the performance of each program is evaluated based on the running time on 200 iterations. The code related to plotting was removed in evaluation, since plotting through Xlib is time-consuming which could weaken the difference between different implementations. To analyze the performance of the programs, they were run on a cluster with three different data size - 100, 500, 2500 and with 1 - 16 processes/threads respectively. For MPI+OpenMP version, the number of MPI processes was fixed as 4.

### Sequential version analysis

The running time of the sequential program is shown as below:

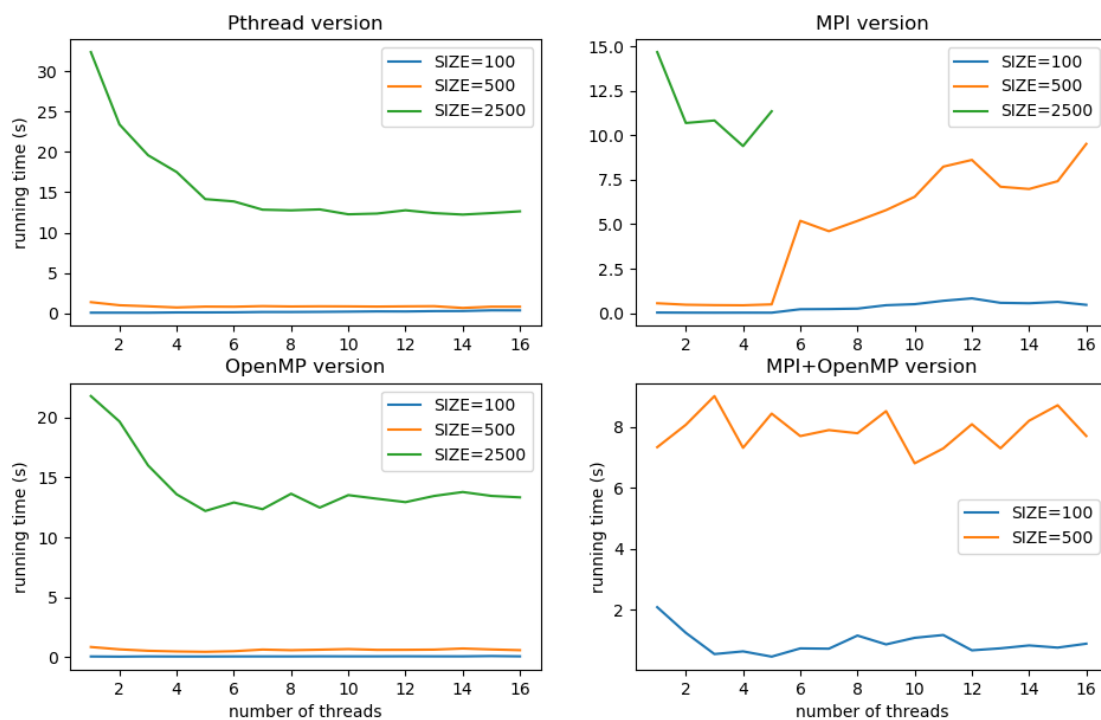
Window size	Running time (s)
$100 \times 100$	0.045720
$500 \times 500$	0.952286
$2500 \times 2500$	19.938146

More data points were generated with different data sizes:



Observing from this graph, the running time is small when window size is small, but it increases fast as data size increases. In fact, according to the program design, the complexity of this program is  $O(n^2)$ . This could cause the extremely long running time when the data size become very large. Therefore, parallelization for this program could be effective.

### Performance on parallelization



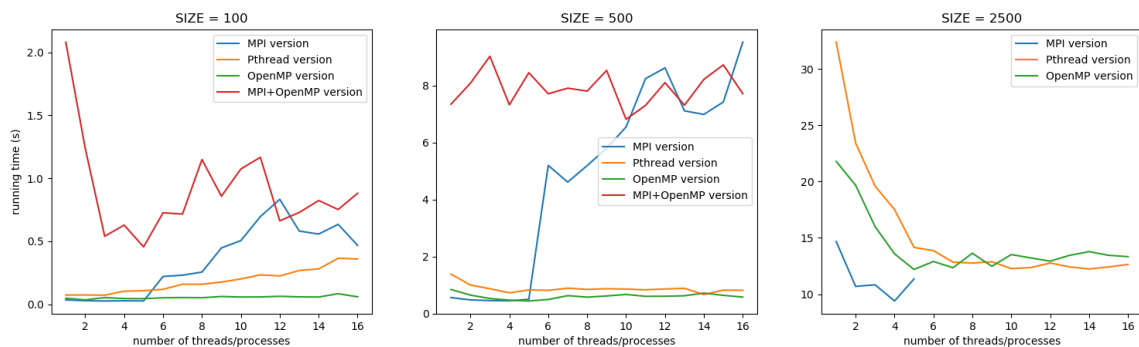
The graphs above show the running time against the number of threads/processes with 3 different data size for each parallel version, which could give an insight of how well the parallelization performed with different processes/threads. Observing the graphs, we could find that:

- For Pthread version and OpenMP version, with large data size (2500), the running time is roughly decreasing as the number of threads increases. This does make sense that multithreads divided the task so that the running speed was increased.
- According to the graph of MPI version, there is no obvious speedup when it was parallelization. In contrary, with data size  $500 \times 500$ , when the number of processes become larger than 5, the running time had a sharp increase. Note that the data points for large data size were missing when the number of processes is larger than five. This is because it was timeout (more than 60 seconds) when the program run on the cluster.

Therefore, we could said that the MPI version performed bad in this evaluation when it is parallelized to too many processes. This might due to that the first virtual machine was assigned with only 5 physical cores so that it had a great communication cost when more processes were required. This shows that the communication between VMs might be time-consuming and it also indicates that the code should not be responsible for this bad performance.

- For MPI+OpenMP version, there is also not obvious speed up and all the data points for large data size were missing. Based on the analysis above, this might be caused by the inefficiency of MPI in this case. It worked even worse than pure MPI version. Note that for MPI+OpenMP version, the number of processes set for MPI is 4. The communication cost was magnified so that the program had a very bad performance.

### Pthread vs MPI vs OpenMP vs MPI+OpenMP



Observing from the graphs above, we could find that:

- When data size is small (100) or medium (500):
  - The OpenMP implementation had the best performance, and it was stable whatever the number of threads changed.
  - The Pthread implementation also performed well but when data size is small, it becomes slow when more threads are required. This might also be because of the communication cost inside pthreads.
  - The MPI version performs very well, and even better than all other versions, when the number of processes is smaller than 5. However, its running time becomes unacceptably high when more processes are required, which has been explained before.
  - The MPI+OpenMP version had the worst performance in these two cases; great communication cost for both parallel methods might be the blame for it.
- When data size is large (2500):
  - The OpenMP implementation performed better than Pthread version when the number of threads is smaller than 5 but came to the same level when more threads were used.
  - With no more than 5 processes, the MPI version did really well and beat the other two versions. The MPI model could be said as the best parallel implementation if the cost between machines was not taken into account.

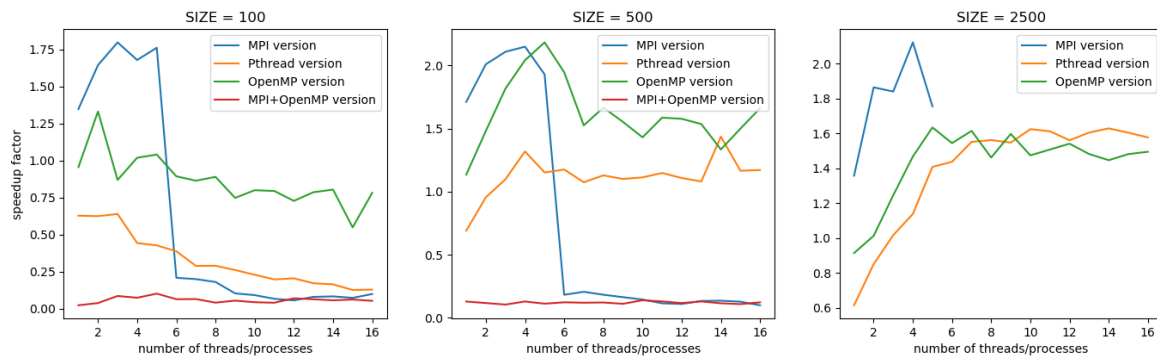
## Sequential vs Parallel

In the comparison between parallel version and sequential version, *speedup factor* could give a clear insight about how the parallel computation performs, which measures the increase in speed by using multiprocessors.

The **speedup factor** is given as:

$$S(n) = \frac{t_s}{t_p} \quad (2)$$

, where  $t_s$  is the execution time on a single processor (sequential) and  $t_p$  is the execution time on a multiprocessor. Therefore, the speedup factors for different numbers of threads/processes with different data size could be computed, and they were plotted as follows:



According to the graph, we could find that:

- When data size is small, the speedup is not obvious, and in some cases the parallel programs perform even worse than sequential one. Adopting more threads/processes caused worse performance, since computation work was not heavy so that the communication time dominated the running time.
- When data size becomes large, the speedup becomes obvious. Moreover, with heavy computation work, the communication time was flattened so for Pthread and OpenMP, it kept better performance than sequential one, which indicates the advantage of parallelization.

## Conclusion

In this experiment, five implementations of Heat Simulation - Sequential version, MPI version, Pthread version, OpenMP version and MPI+OpenMP version have been successfully designed and tested. The performance analysis on these five programs has been presented. The parallel computation does a good job on speedup when the data size is large. Especially for MPI with less than 5 processes. However, due to the limitation of cores in VMs, the MPI implementation works badly when more processes are taken. And for OpenMP+MPI implementation, its performance was bad. This might be due to some bad design since this combination is supposed to be the best performer on large size of data and multicores. Therefore, some investigations are needed for future improvement.