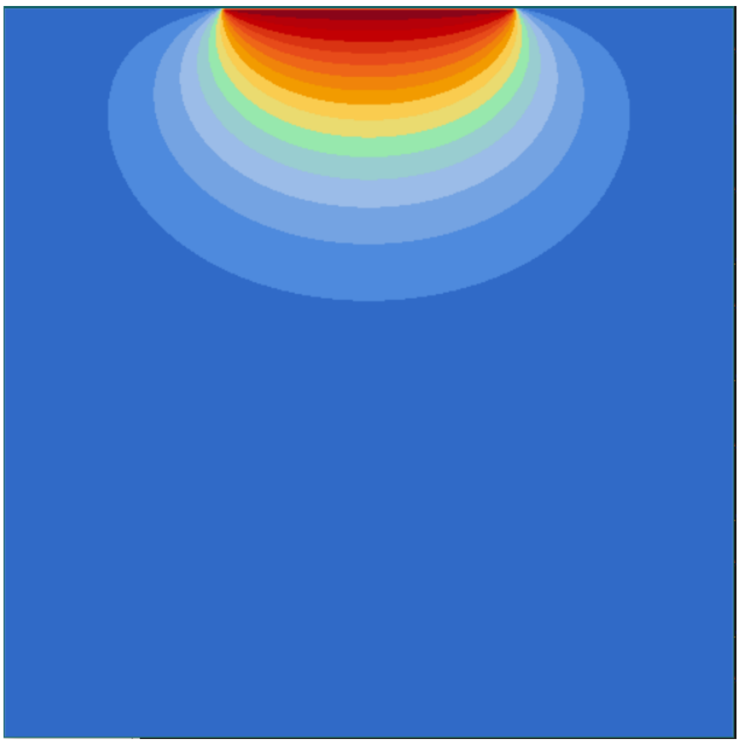
CSC4005 HW4: Heat Simulation Report

Min Tian 116010168



Content

1. Introduction

1.1 Jacobi Iteration.

Jacobi Iteration is invented to solve a $n \times n$ linear equations set. For a given matrix *A*, it can be decomposed into a diagonal component *D*, and the remainder *R: A = D + R*

The solution is then can be obtained iteratively via:

$$x^{k+1} = D^{-1}(b - R\mathrm{x}^{k})$$

Where x^{k} is the kth approximation, or iteration, of x and $\mathrm{x}^{\mathrm{k+1}}$ is the next, or k + 1, iteration of x. the formula is shown below:

$$x_i^{k+1} = \frac{1}{a_{ii}}\left(b_i - \sum_{j \neq i} a_{ij} x_j^k\right), i = 1,2,\ldots,n$$

The standard convergence condition is when the spectral radius of the iteration matrix is less than 1: $\rho(D^{-1}R) < 1$. A sufficient but not necessary condition for the method to converge is the matrix A is strictly or irreducibly diagonally dominant. Strict row diagonal dominance means that for each row, the absolute value of the diagonal term is greater than the sum of absolute values of other terms:

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}|$$

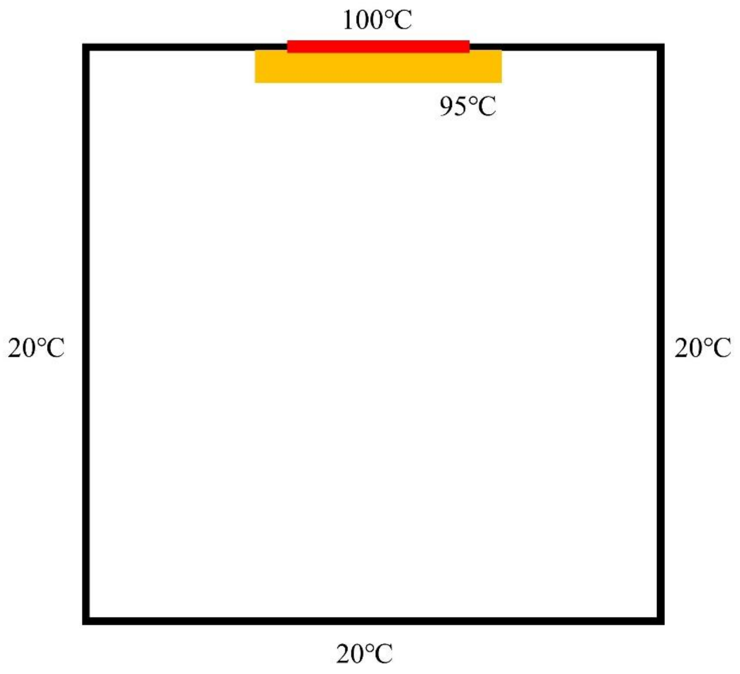
1.2 Heat Diffusion on 2D plane

Dividing an area into mesh of points, $h_{i,j}$. The temperature at the inside point can be taken to be the average of the temperatures of the four neighboring points.

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

1.3 Goal

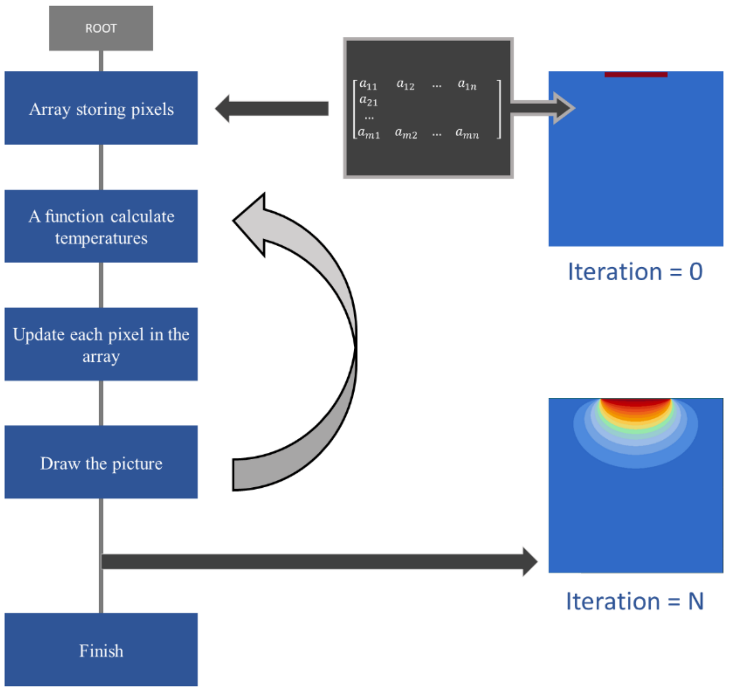
Simulate a room with four walls and a fireplace. The temperature of the wall is 20℃, the temperature of the fireplace is 100. Write sequential, MPI, Pthread and OpenMP version of the program. Plot the temperature contours at 5℃ intervals using Xlib. (The boundaries of the colors are just a demonstration).



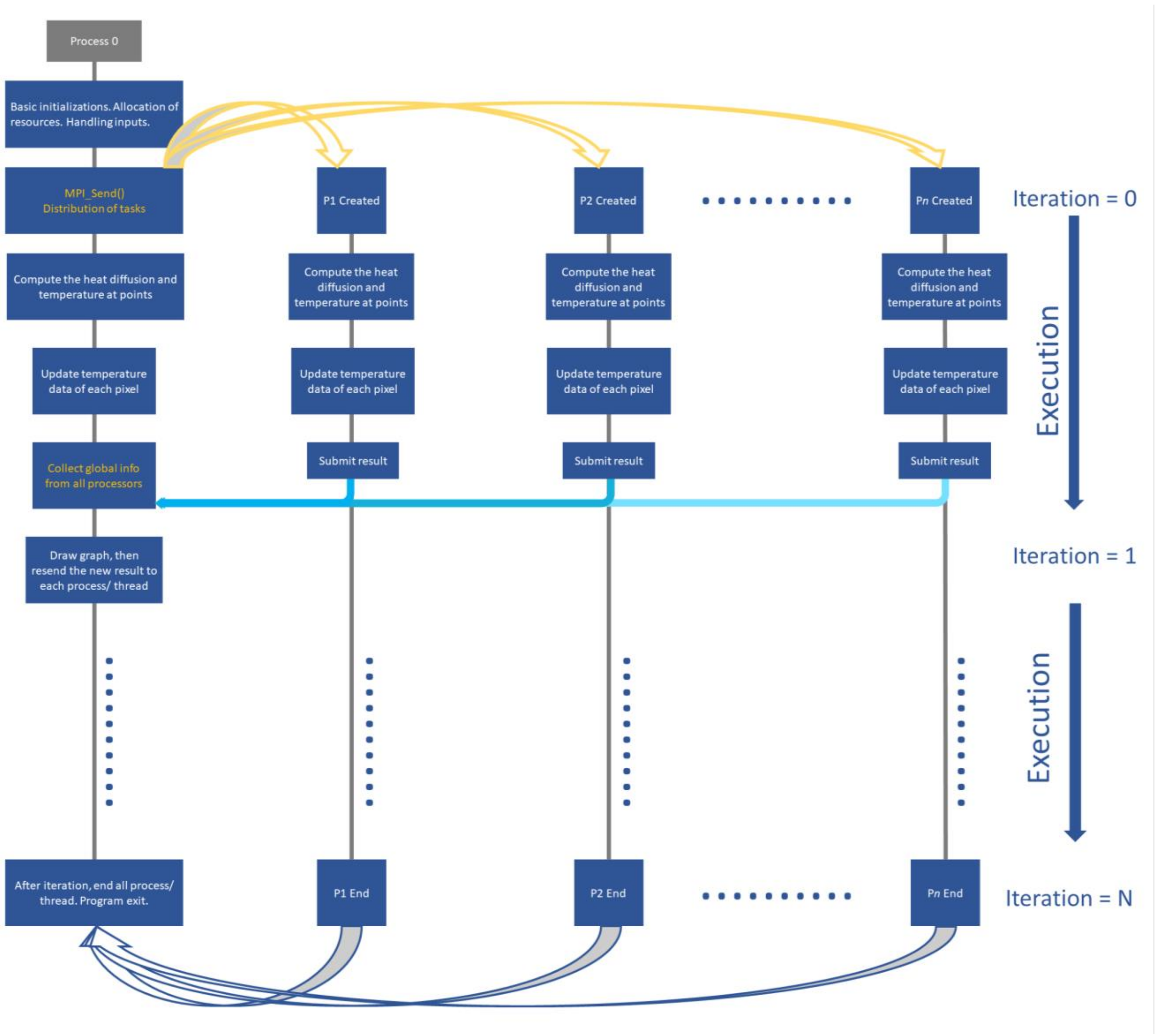
2. Design

2.1 Program Flow

In this part, I'll use diagrams to illustrate the sequential and parallel version's execution flow. First is the sequential version:


ROOT
Array storing pixels
A function calculate temperatures
Update each pixel in the array
Draw the picture
Finish
a_{11} a_{12} ... a_{1n}
a_{21}
...
a_{m1} a_{m2} ... a_{mn}
Iteration = 0
Iteration = N

The sequential execution is quite straight-forward. The MPI version diagram is shown as follows. Because the Pthread and OpenMP version are technically quite similar, I won't draw extra diagrams for them. However, I will write down some tricky places, which need to pay special attention on.


Process 0
Basic initializations. Allocation of resources. Handling inputs.
MPI_Send() Distribution of tasks
P1 Created
P2 Created
Pn Created
Iteration = 0
Compute the heat diffusion and temperature at points
Update temperature data of each pixel
Collect global info from all processors
Submit result
Execution
Iteration = 1
Draw graph, then resend the new result to each process/ thread
Execution
After iteration, end all process/ thread. Program exit.
P1 End
P2 End
Pn End
Iteration = N

We have experience from homework 3 that storing data in 2-D array when using MPI may cause some troubles. The reason behind this is that 2-D array are not actually stored in a contiguous way in the memory. This data structure can vary from machine to machine. In order to prevent things like this happen, we nee to manually define a new MPI_Type. Once you done with the declaration, you can pass the data whichever way you like.

As shown in lecture slides, the difference between strip partition and block partition both has their own advantages and drawbacks. For block partition, when for edges where data points are exchanged, communication time is

$$t_{commsq} = 8\left(t_{startup} + \sqrt{\left(\frac{n}{p}\right)}\, t_{data}\right)$$

For those two edges where data points are exchanged. The strip partition communication time is given by

$$t_{commcol} = 4\left(t_{startup} + \sqrt{n t_{data}}\right)$$

In general cases the strip partition has a better performance for large startup time, and a block partition has advantage for a small startup time.

Another trick is work partition. I've implemented this method in homework 2, the Mandelbrot Set one. Because the height or width of the canvas may cannot be fully divided by the number of process or thread, we could add an extra chunk, but not necessarily to compute it, by testing the extra space with a function.

2.2 Timing

Time measurement result in parallel programming can be vary greatly between different measurement methods. To avoid this problem, *MPI_Wtime()* and *clock_gettime()* are used instead of *clock()*. Because *clock()* is the timing function for the CPU running time rather than the program execution time. The result measure by it will significantly larger than the actual value due to the fact that *clock()* takes the sum of all CPU's running time into account.

3. Program Execution

3.1 Testing Environment

5 Intel(R) Xeon(R) CPU E5−2699 v4 @ **2.20GHz**

{1 physical id : 0; 1 address sizes : 46 bits physical, 48 bits virtual} ×**5**

3.2 Programs Execution Command

Compile Files:

```
make all
```

For Pthread version:

Parameters: **width, height, iteration, #threads, enable(1)/disable(0)Xlib**

```
./pthread 200 200 800 4 1
```

For MPI Version:

Parameters: **width, height, iteration, enable(1)/disable(0)Xlib**

```
mpirun -np 4 ./mpi 200 200 800 1
```

For OpenMP Version:

Parameters: **width, height, iteration, #threads, enable(1)/disable(0)Xlib**

```
./omp 200 200 800 4 1
```

For Sequential Version:

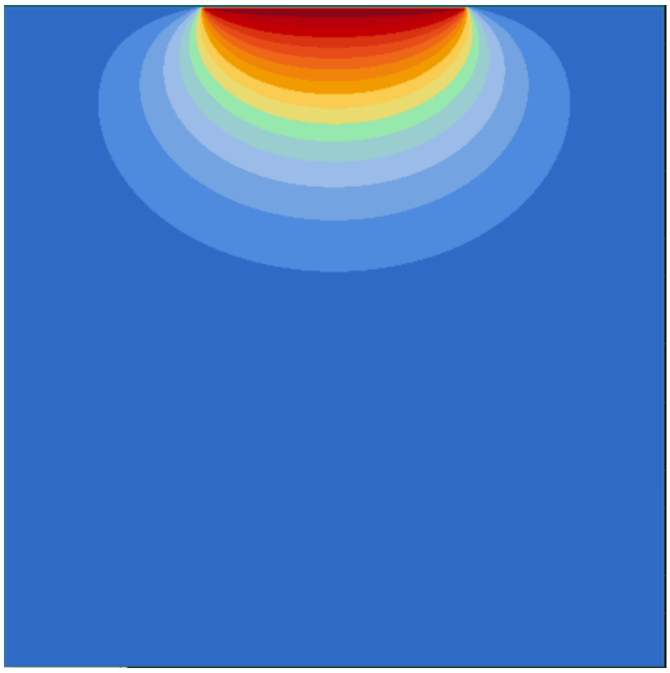
Parameters: **width, height, iteration, enable(1)/disable(0)Xlib**

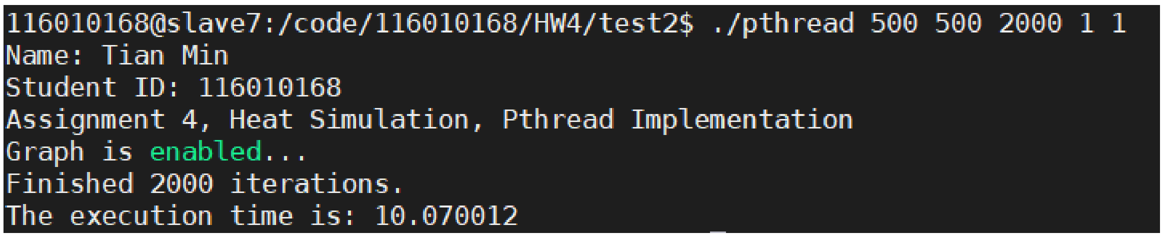
```
./seq 200 200 800 1
```

Clean

```
make clean
```

3.3 Expected Result





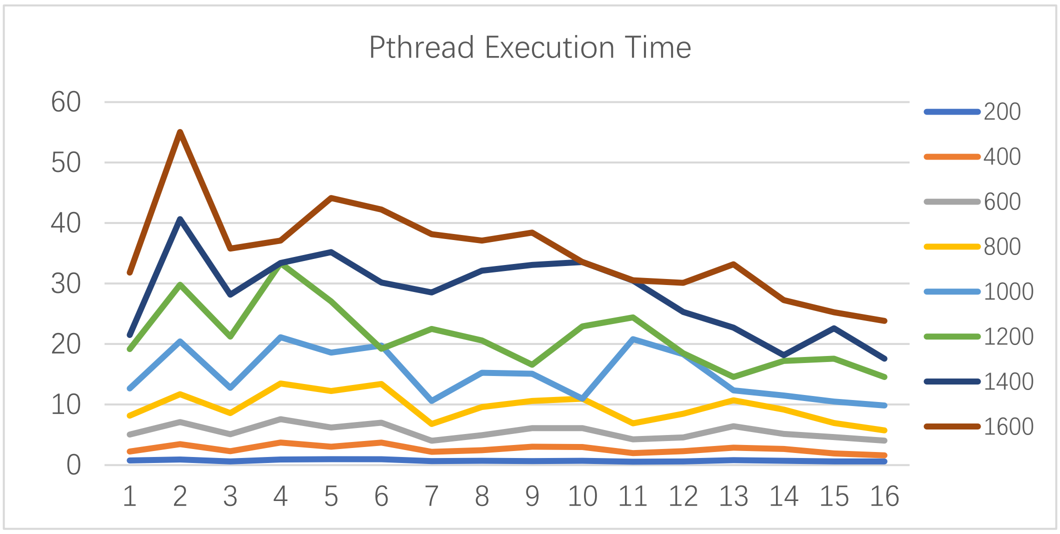
```
116010168@slave7:/code/116010168/HW4/test2$ ./pthread 500 500 2000 1 1
Name: Tian Min
Student ID: 116010168
Assignment 4, Heat Simulation, Pthread Implementation
Graph is enabled...
Finished 2000 iterations.
The execution time is: 10.070012
```

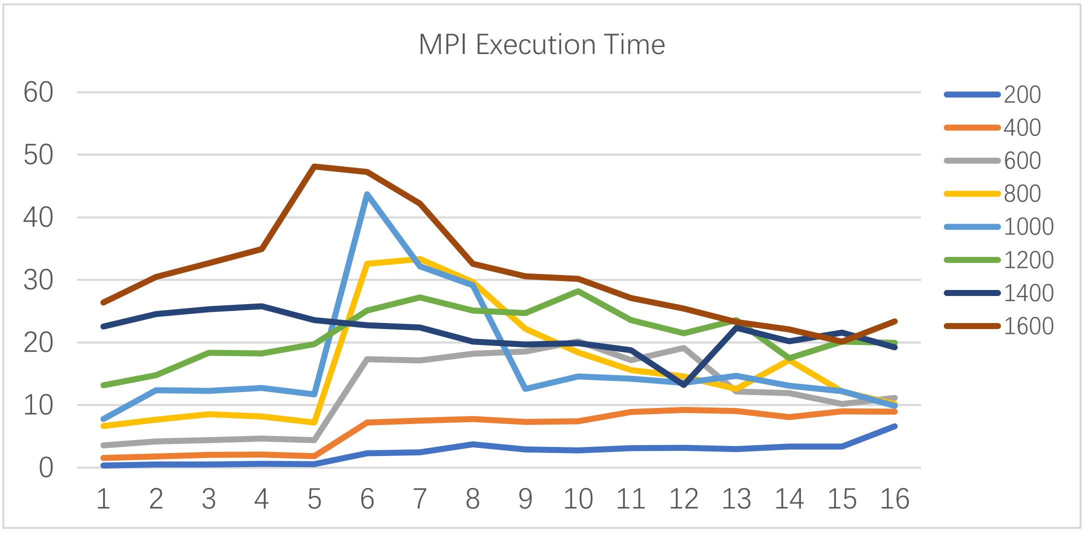
Terminal output would also contain the assignment information and the running time. We can clearly see from the image that temperature gradually diffuses from the centre of the fireplace. There were some little flickers during the image generation on my local environment that I cannot explain. The result should be fine on the sever environment as displayed in the video.

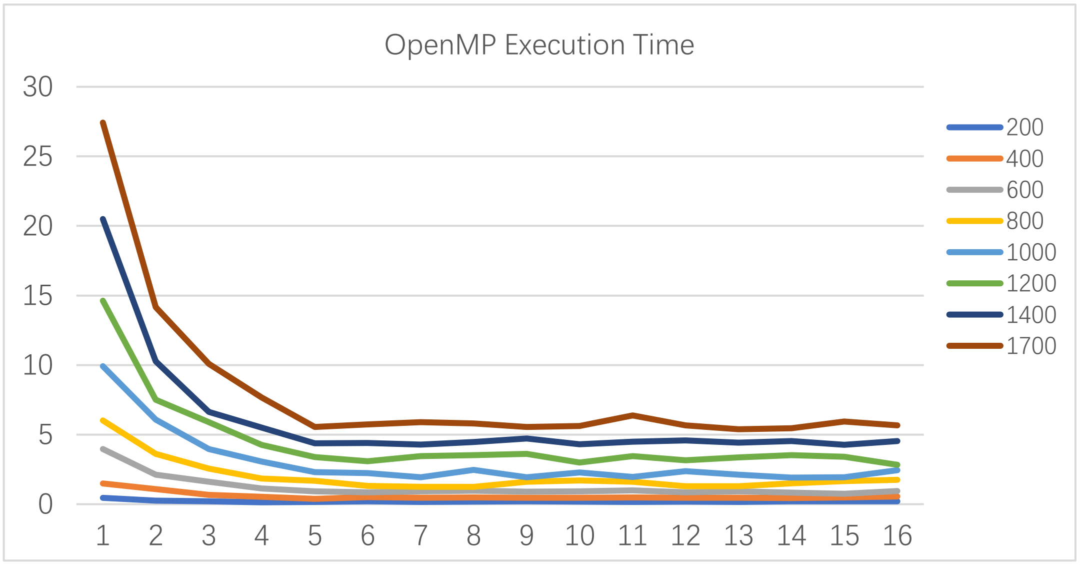
4. Performance Analysis

4.1 Execution Time

The experiment was designed as follow: 1) Set a fix iteration number of **800**. 2) Run MPI, Pthread, OpenMP program separately. 3) Record the execution time of process/ thread number from 1 to 6. 4) Gradually increase the size of the room from 200×200, 400×400… to 1600×1600. The graph below shows the examination outcome.

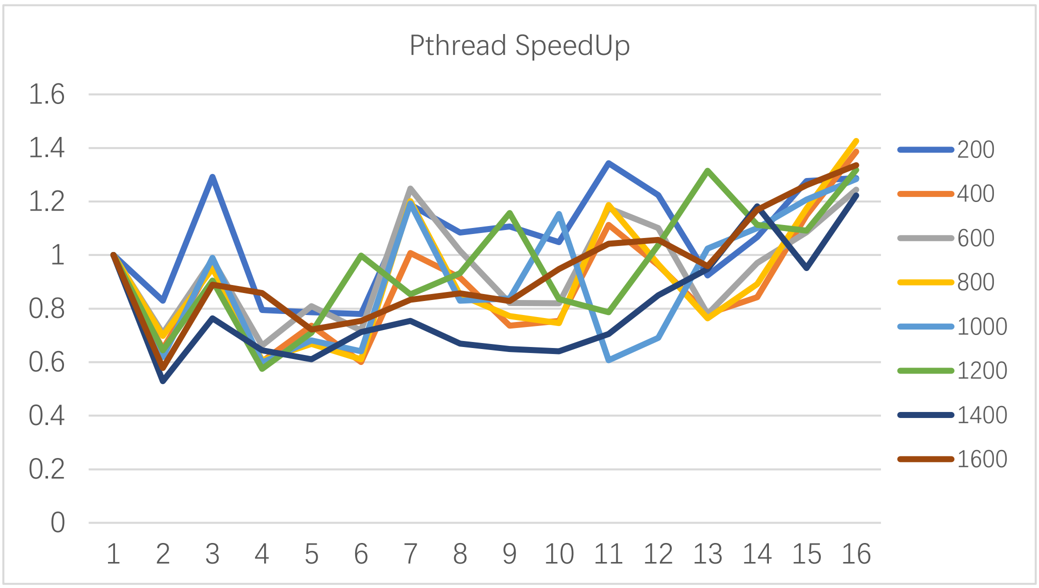


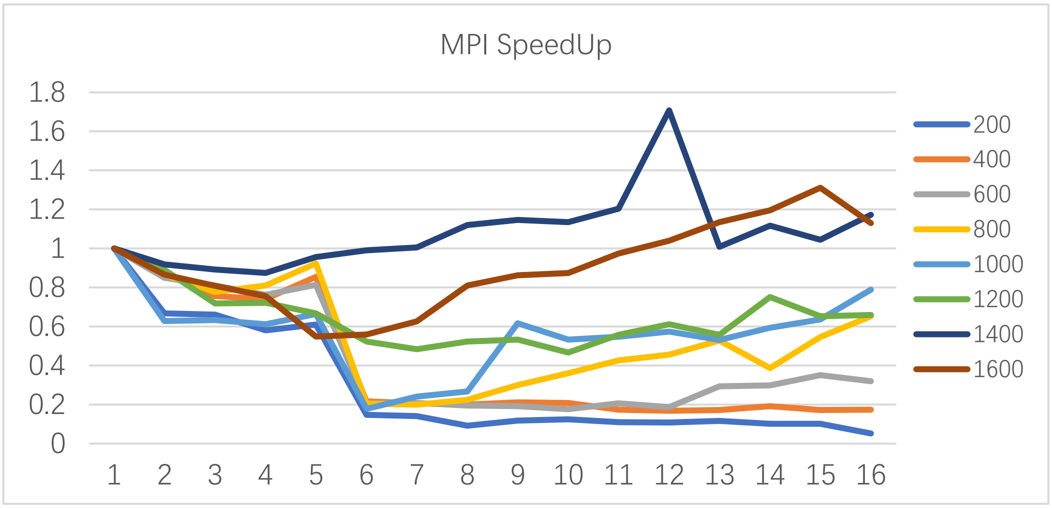


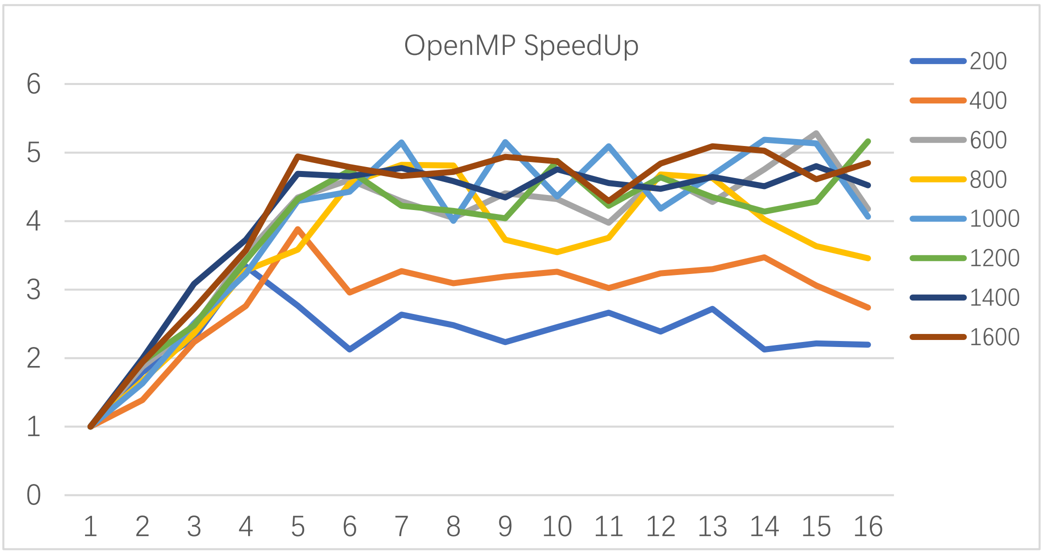


4.2 Speedup Factor

According to definition, we can calculate the speedup factor by $S(n) = \frac{Sequential\ Time}{Parallel\ Time\ with\ n\ processor}$

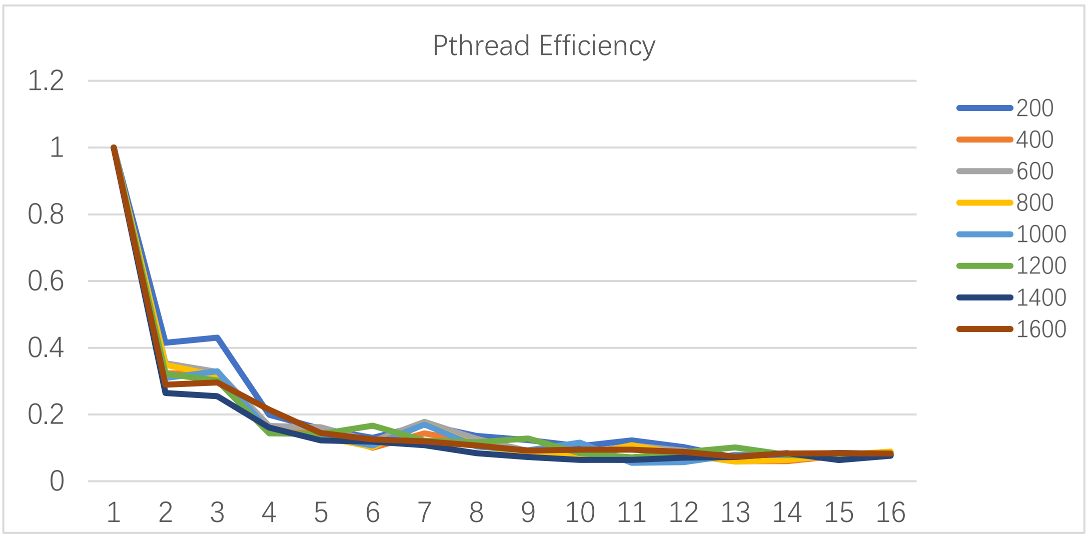


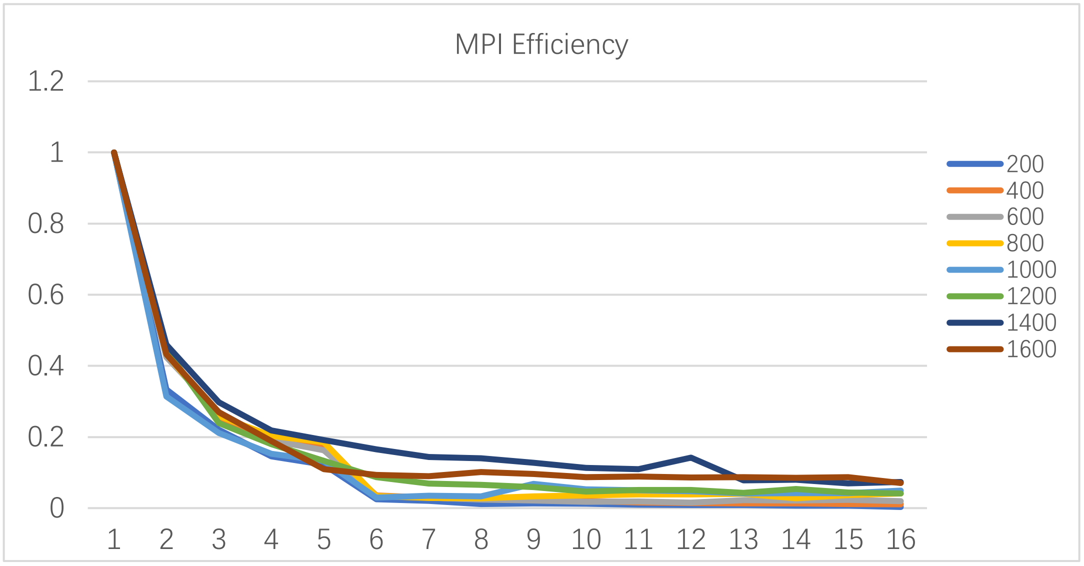


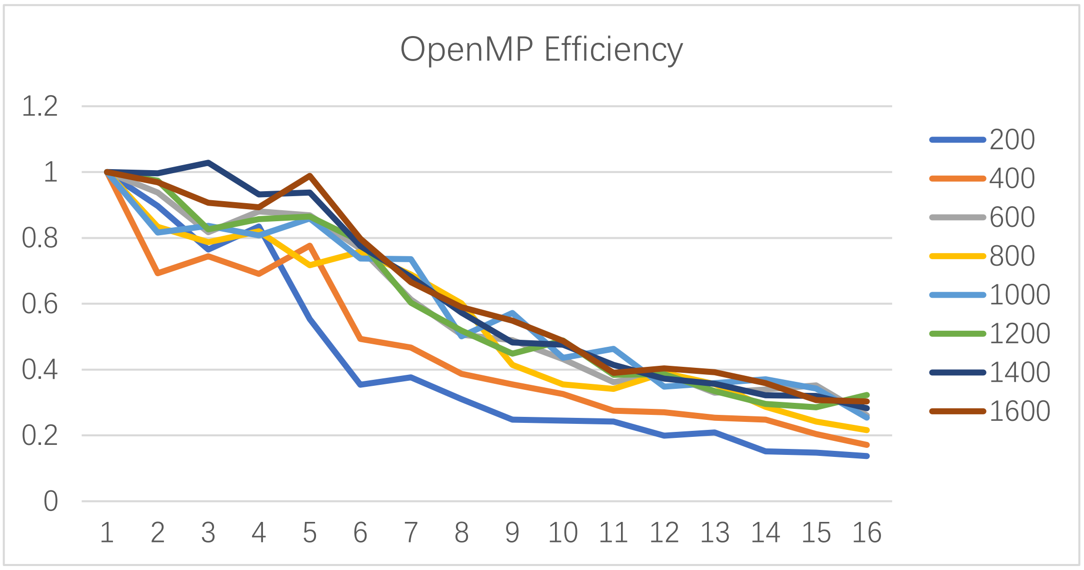


4.3 Efficiency

$$Efficiency = \frac{Execution\ Time\ with\ one\ processor}{Execution\ Time\ with\ multi{-}processor \times \#processor} = \frac{T_s}{T_p \times n} = \frac{S(n)}{n} \times 100\%.$$







4.4 Analysis

As the results shown above, MPI communication take up a great partition of execution as test size increases. Also, it needs more scheduling time, as the number of processes increases. In the graph of efficiency, it clearly shows that as the number of processors goes up, efficiency drops steeply. One thing confused me is the execution time of the MPI version. There's a "hill" in the lines. First the execution time went up to some maximum and then went down. This can be explained by some accidental events or the speedup gradually outweigh the communication overhead.

For the Pthread version, its execution time table shows a serration contour as expected from the previous homework. Pthread version tests have a speedup table with great variations. The efficiency drops quickly when the number of threads increase from 1 to 3. It stabilizes at somewhere lower than 10%, which is not a very good result, and close to MPI's.

As for the OpenMP, it reached the best performance in this experiment. Although there is not much improvement after thread number larger or equal to 5. Under most situations, OpenMP version can get a speedup factor larger than 4. Its efficiency can stay at 37% when the thread number is increased to 16.

5. Conclusion & Improvement

5.1 Conclusion

This homework is a practical exercise on parallel programming. I think the main difference between this one and the previous ones are the potential improvement parts and the data dependency. Heat simulation has lots of potential improvements. For example, the distribution is symmetric, we could have done have the computation, or we can spare more space storing only left or right half of the plane. Data dependency I'm talking about is that, when simulating heat distribution, we need pixels around's message to determine one pixel. This cause troubles like, we need to preserve the map data, however it's sometimes too big for our RAM to hold. In this situation, we need to do things smart like, only passing the edge data to each process.

5.2 Incremental Strategy

Saw this strategy in *Jianfeng CHEN*'s work, due to the time limitation I didn't implement it. But it's brilliant, it's graceful, will take a look into that code later. The core idea of this method is: since the room is big, why not shrink it small, then calculate, and put it back to original size? This method should work with a special design convergence threshold, which helps to adjust the accuracy after resizing map back.

6. Reference

[1] *Prof. Yeh-Ching CHUNG: http://www.cs.nthu.edu.tw/~ychung/slides/para_programming/slides-06.pptx*

[2] *Jianfeng CHEN: https://github.com/cjf00000/Heat-Distribution*

[3] *Temperature to RGB color: https://wenku.baidu.com/view/4e827f3f5727a5e9856a61cb*

[4] *Wikipedia Jacobi Iteration: https://en.wikipedia.org/wiki/Jacobi_method*

[5] *Welkin Reardon: https://welkin.dev/2019/10/16/CLOCK-MONOTONIC/*

[6] Cnstz: *https://github.com/cnstz/heatTransferSimulation-parallel*

[7] Intel OpenMP Tutorial: *https://www.youtube.com/watch?v=jdYHVeh2wEI*