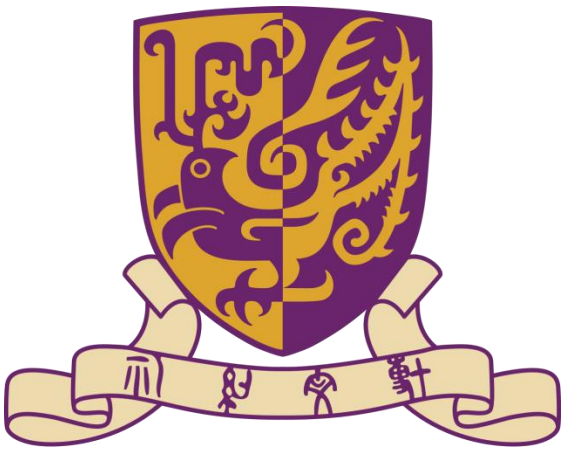


THE CNINESE UNIVERSITY OF HONG KONG, SHENZHEN

CSC4005

Distributed and Parallel Computing

---

# Report for Assignment 4
# Heat Simulation

---


*Author:*
Li Jingyu 李璟瑜

*Student Number:*
118010141


Dec 6, 2021

# Contents

# 1 Introduction

Assignment 4 requires us to design a program for Heat simulation and render the points on screen with a type of GUI. In a two-dimension square room, there are four walls and a fireplace. The walls have certain temperature and the fire source emits the highest heat to warm this room. Different temperature represents different color, with red means high and blue means low.

The heat will spread to the neighbor space as time goes by. Here we use Jacobi iteration to simulate the heat distribution. In each time elapse, the temperature of the current point is simply the average of the four neighboring points. When the difference of previous temperature value and the current one reaches a critical value, tolerance, the iteration will stop and the room becomes stabilized.

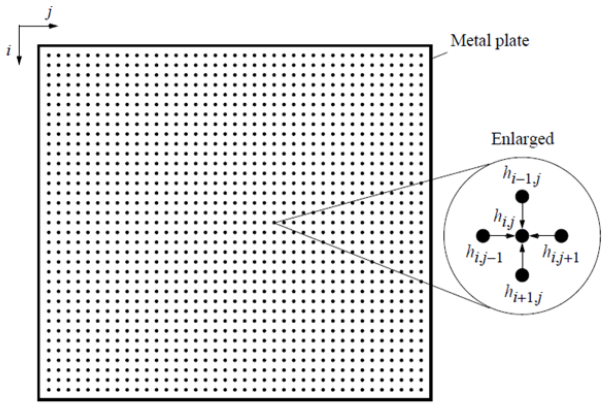



Figure 6.11 Heat distribution problem.

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

Our task in the assignment is to keep computing the temperature of each point in the room within a certain elapse and rendering the heat state in the screen using ImGui (a C++ supported graphical user interface) and explore the relationship between **computational speed** and different parameters.

The sequential version code is given on blackboard while we should attempt to implement the parallel versions using **MPI** and **Pthread** and **openMP** and **CUDA** approaches. The combination of **MPI & openMP** version serves as the bonus.

MPI (Massage Passing Interface) is the standard of message passing function library that includes many interfaces. It is a parallel technology based on the communication between processes. A sample MPI interface is shown as follows. MPI_Send is one of the most commonly used functions in MPI.

```
MPI_Send(
    void* data,
    int count,
    MPI_Datatype datatype,
    int destination,
    int tag,
    MPI_Comm communicator);
```

```
int pthread_create(
    pthread_t *restrict thread,
    const pthread_attr_t *restrict attr,
    void *(*start_routine)(void *),
    void *restrict arg);
```

Pthread refers to POSIX threads, which is an execution model that exists independently from a language and a parallel execution model. It is primitive and practical in C programming utilizing the lightweight threads. Here shows the basic grammar of creating a thread and synchronizing the thread.

```
int pthread_join(pthread_t thread, void** return_value);
```

OpenMP is an application programming interface that supports multi-platform *shared-memory* multiprocessing programming. Only certain #pragma is required to add before the codes to realize parallel programming. No extra initialization is needed. It is probably one of the most convenient approaches in code effort among the four.

CUDA refers to Compute Unified Device Architecture. It is a parallel computing platform and application interface that utilizes the power of GPU. Only NVIDIA GPU cards are supported. The CUDA approach includes many complex concepts including signs (device, host, global), memory organization (grid and block and thread), memory management interfaces, and so on. Here shows a kernel function to be executed by the GPU.

```
kernal_name<<<grid_size, block_size>>>(args);
```

Although the parallel computing approaches may be different, the core part of this assignment is the same, which is to partition the room space calculation reasonably and allocate to different processes/threads. Proper synchronization and data aggregation strategies are also required. Successful coding renders careful consideration and practice necessary.

# 2 Design

In this part the logic of the codes and some details will be elaborated. All of the implementations are based on the ImGui template on blackboard.

- Sequential part

I will first briefly introduce the sequential part and some general configuration and codes in the programs which will be executed by all the different versions.

In the header file, a **grid** is defined as a structure having two vector data fields representing each point's temperature in the room. The reason why we implement two vectors is to separate previous room state (for reading) and next room state (for writing), which proves to be a fantastic strategy to avoid data race. Besides, grid structure also owns an identifier to decide which is the current buffer and the length of room. Another structure **state** is used to store the parameters like room size, position of fire source, temperature value, tolerance, whose values are initialized to constants.

In computation part, first choose the appropriate algorithm (either Jacobi or Sort). Then we just simply use a double for loop to iterate all the points in the room, update their temperature by averaging four neighboring points, test whether the difference is smaller than tolerance, and write the new value to a different buffer. Remember to switch the current buffer to another one at last. In this way we finish an elapse.

In main function, current state with many parameters for this problem and the grid representing the room are defined. Graphical engine is initialized. In each iteration we render the temperature of points with different colors after calculation. By repeated computation and updating on temperature, it generates a Heat simulation.

- MPI version

In MPI programming, the programmer should attach importance to the message passing between processes.

In the main function, initialize the MPI setting first. Initialize the state and grid. These steps are expected to do be done in each process.

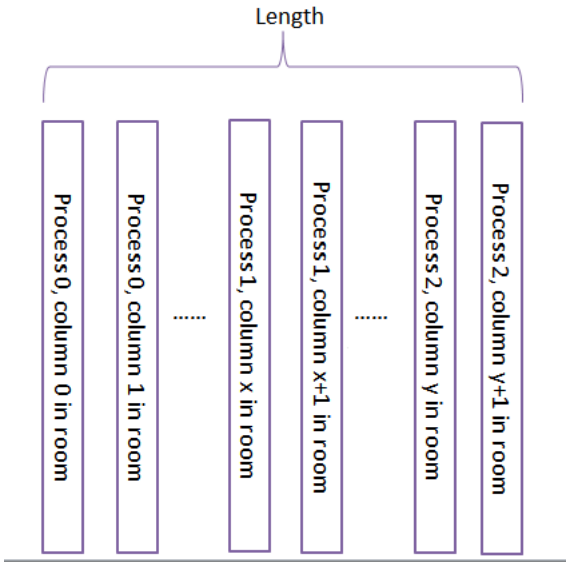
In the master process, set up the graphical context for plotting. The rendering function will run repeatedly in order to stabilize the room and obtain more average and accurate speed. Set a dragger for adjusting the blocking size dynamically in the run time to gather a better view on the room. However, the procedure is usually slow, especially in the server. So it would better fix the parameters in advance. For other parameters like room size and temperature, I eliminate the dragger since it will cause unnecessary data re-initialize and process synchronization work.

The following part will be treated as the **computation cost** (time). *We only care about computation time in this project*, since drawing costs hugely in the GUI rendering context that is often done with a single process/thread.

In the calculation, we already have the parameters (meta-data) stored in state initialized in each process, so we only need to broadcast the current buffer's data to all the processes in MPI_COMM_WORLD. In this way we inform all salve processes the previous state of the temperature, and they can deal with the part they are responsible for. Also, I do not partition the data in advance but send all the data in order to eliminate the complex issue of data dependency. The method of get_current_buffer() provides the convenience for choosing the current buffer.

```
// Broadcast grid
MPI_Bcast(grid.get_current_buffer().data(), grid.length * grid.length, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Enter the *calculate* function. In the computation part, I divide the whole room into several columns group decided by the index i, in the x-coordinate. In this way the division in the double for loop becomes easier.




I calculate the **start and end indexes** of the columns this process will deal with and execute the for loop accordingly.

```
int task = grid.length / grid.proc_num;
int remain = grid.length % grid.proc_num;  // extra columns
size_t start = grid.rank * task + remain;
size_t end = start + task;
```

Because the room length may not be divisible by the process number, it may lead to extra columns to be computed. I leave the work to the master process. For the first *remain* columns, they are distributed to master process. Using the *start* and *end* indexes, the process can continue their work. After the single point's temperature has been computed, the result is written to another buffer, separating the read and write procedure. Remember to switch the buffer at last!

```
if (grid.rank == 0) {
    start = 0;
}
```

```
for (size_t i = start; i < end; ++i) {
    for (size_t j = 0; j < state.room_size; ++j) {
        auto result = update_single(i, j, grid, state);
        stabilized &= result.stable;
        grid[{alt, i, j}] = result.temp;
    }
}
grid.switch_buffer();
```

After we evenly distributed the columns to each process, we use "*MPI_Gather*" to gather the all results in different processes. **Only indexes between start and end take effects**!

```
MPI_Gather(&grid[{rank * task + remain, 0}], task * grid.length, MPI_DOUBLE,
&grid[{remain, 0}], task * grid.length, MPI_DOUBLE, MASTER,
MPI_COMM_WORLD);
```

In each single calculation, a Boolean variable is used to record whether the current point reaches the stable state. Since we utilize many processes to finish the task, we also need to gather all the Boolean values and **&** then.

```
bool finishedSlave = false;
for (int p = 1; p < proc_num; p++) {
    MPI_Recv(&finishedSlave, 1, MPI_CHAR, p, MASTER, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    finished &= finishedSlave;
}
```

The calculation part then finishes.

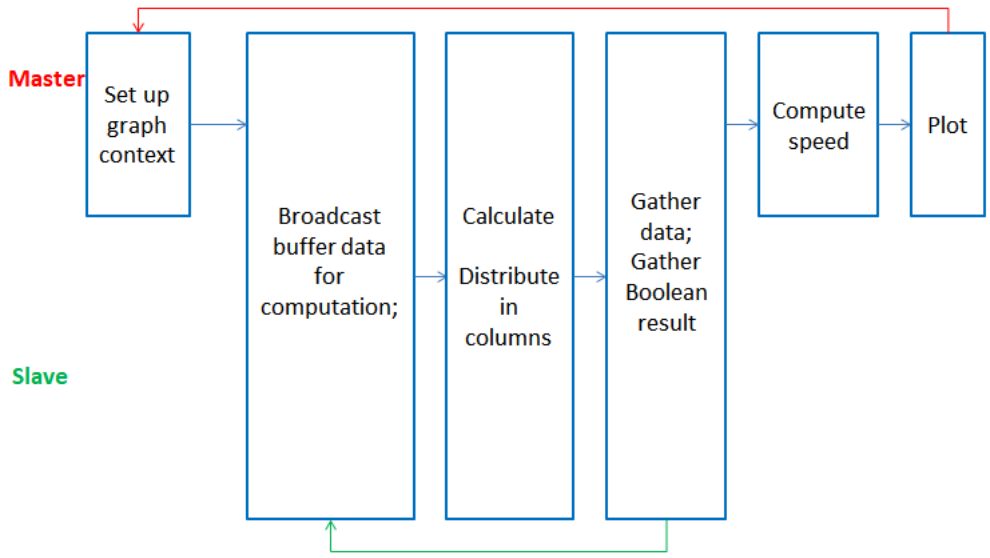
In slave process, the procedure is similar, receive broadcast of current buffer's data in grid from master process, perform calculation and return back the data part they are responsible for as well as the Boolean result. Repeat the procedure using a while loop.

In the main process, the computation speed will be calculated, whose value equals to the processed iteration (# of elapses) divided by the duration (in seconds). The graph is then plotted.

```
count++;
duration += duration_cast<std::chrono::nanoseconds>(endCal - beginCal).count();
if (count % ITERATION == 0) {
    std::cout << ITERATION << " elapse with " << duration << " nanoseconds\n";
    double speed = double(ITERATION) / double(duration) * 1e9;
    std::cout << "speed: " << speed << " iterations per second" << std::endl;
    duration = 0;
}
```

The master process will continue to run the computation function, calculate time cost and plot the graph. The slave process will repeat the computation part. After the room temperature is stabilized, stop computation but continue to render the graph.

A structural flow figure is displayed.




- Pthread version

In Pthread programming, the programmer should lay more emphasis on thread synchronization and avoid dead lock and data race.

Compared with MPI version, Pthread is similar but easier to implement. All the threads are able to read and write the global *grid* data directly since they share the same memory space. The state storing all parameters for calculation and the Boolean results *finished* are initialized as global variables to be visible in all threads. To represent different thread, I use an **atomic integer** as the thread number (rank in MPI version), which can avoid deadlock, ensure thread safety and provide better performance than *mutex*. The design of two buffers separates read and write and smartly realizes **data race prevention**.

```
// Global variables
static hdist::State current_state;
auto grid = hdist::Grid{
    static_cast<size_t>(current_state.room_size),
    current_state.border_temp,
    current_state.source_temp,
    static_cast<size_t>(current_state.source_x),
    static_cast<size_t>(current_state.source_y)
};
std::atomic_bool finished{false};
```

In the main function, the program receives thread number from command line. Similarly, the main thread will set up graphical context. In the calculation part, create threads and execute **a wrapper function** of the calculation function. Join threads afterwards.

```
/* Start calculation */
auto beginCal = std::chrono::high_resolution_clock::now();

current_thread = 0;
finished = true;
for (int i = 0; i < thread_num; i++) {
    pthread_create(&comp_threads[i], nullptr, thread_calculate, nullptr);
}
for (int i = 0; i < thread_num; i++) {
    pthread_join(comp_threads[i], nullptr);
}

grid.switch_buffer();

auto endCal = std::chrono::high_resolution_clock::now();
/* Finish calculation */
```

```
void *thread_calculate(void *) {
    bool thread_finished = hdist::calculate(current_state, grid);
    if (!thread_finished) finished = false;
    return nullptr;
}
```

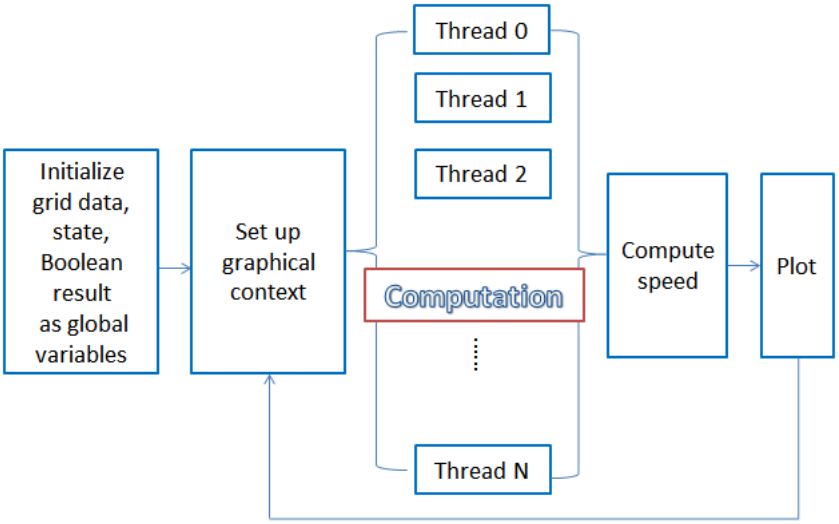
Inside the computation function, one of the tricky parts is to **avoid data race**. A data race happens when two threads try accessing the same position in the memory. For example, when one thread reads a value of position for calculation, another thread writes to the position value immediately, before the first thread can obtain the result. The common scenarios are **Write after Read**, **Write after Write**. To avoid this, separate the reading and writing is required, as mentioned before. During the computation, the threads read the data from one buffer (previous state) and write to another buffer (new state). After all computation in this step finishes, exchange the two buffers. Another tricky part is that the *switch* function in written inside the *calculate* function at first, but all the threads will execute the function and operate on the global grid, exchange the buffer again and again, causing the bugs. So the *switch* function needs to be extracted to *main* function.

The data partition logic is similar with that in MPI, using *start* and *end* indexes. The only difference is that we now use *thread_num* but not *proc_num*. Thread synchronization is not required since no data dependency exists.

```
int task = grid.length / thread_num;  // Total columns
int remain = grid.length % thread_num;  // extra columns
size_t start = rank * task + remain;
size_t end = start + task;
```

```
for (size_t i = start; i < end; ++i) {
    for (size_t j = 0; j < state.room_size; ++j) {
        auto result = update_single(i, j, grid, state);
        stabilized &= result.stable;
        grid[{alt, i, j}] = result.temp;
    }
}
break;
```

The program flow shows as follows.




- OpenMP version

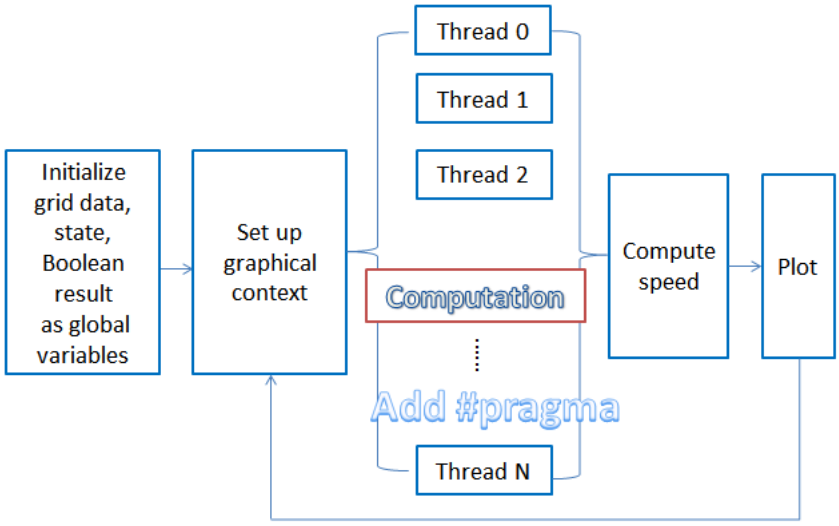
OpenMP version is extremely easy to implement! The main logic of openMP program is nearly the same as the sequential version except for speed calculation and thread number obtained from the command line.

```
omp_set_num_threads(thread_num);
```

In the computation part, programmer only needs to add the #pragma before *for* loop to parallelize. No explicit initialization of processes or threads is even required. We do not need to calculate the *start* and *end* indexes. Here we should define *i* and *j* outside and define *j* as private variable in each thread, since we iterate each row in different threads and the indexes cannot interfere. For the coding effort, only one sentence is added. How amazing it is!

```
size_t i, j;
#pragma omp parallel for private(j)
for (i = 0; i < state.room_size; ++i) {
    for (j = 0; j < state.room_size; ++j) {
        auto result = update_single(i, j, grid, state);
        stabilized &= result.stable;
        grid[{alt, i, j}] = result.temp;
    }
}
grid.switch_buffer();
break;
```

The below shows the program flow of openMP version.




- CUDA version

In the CUDA version, I include all the codes of hdist.hpp into main.cu for better comprenhension data sharing. During the programming, the global data both visible to CPU and GPU provide great convenience for the implementation. The grid data and Boolean result, total thread number are initialized as the **__device__ __managed__**

global data, which can be accessed by both CPU and GPU directly. Parameters are defined in advance for quick initialization.

```
// Parameters
#define ROOMSIZE 300
#define SOURCEX 150
#define SOURCEY 150
#define SOURCETEMP 100
#define BOARDTEMP 36
#define TOLERANCE 0.1
```

```
__device__ __managed__ double data0[ROOMSIZE * ROOMSIZE];
__device__ __managed__ double data1[ROOMSIZE * ROOMSIZE];
__device__ __managed__ bool finished = false;
__device__ __managed__ int thread_num = 1;
__device__ __managed__ bool first_buffer = true;
```

The original *update_single* function in .hpp file are defined as a __device__ function in *main.cu* only executed on the GPU.

```
__device__ double update_single(int i, int j)
```

The major function to be executed by the GPU called from CPU (the host) is signed with __global__. No arguments are required since we already define them in the very beginning of the code.

```
__global__ void calculate()
```

In the computation function, the thread number is obtained by certain dim calculation. Then using the thread number to partition the task, using the start and end indexes, still. The computation is similar with that in **pthread** version. Each loop will operate the **global memory data** directly. Since each thread is responsible for different index range, no data race will happen.

```
__global__ void calculate() {
    int rank = blockIdx.x * blockDim.x + threadIdx.x;
    int task = int(ROOMSIZE) / thread_num;
    int remain = int(ROOMSIZE) % thread_num;
    int start = rank * task + remain;
    int end = start + task;

    double* writeData = (first_buffer) ? data1 : data0;
```

```
for (int i = start; i < end; ++i) {
    for (int j = 0; j < ROOMSIZE; ++j) {
        double temp = update_single(i, j);
        writeData[i * ROOMSIZE + j] = temp;
    }
}
```

Some code modification for CUDA version is performed because we do not have some fancy member functions like get_current_buffer() in GPU now. However, these changes make the code clean and easy to understand.

With the help of __device__ __managed__ global arrays, the computation part is smooth to implement. It is really a gift for CUDA programmer.

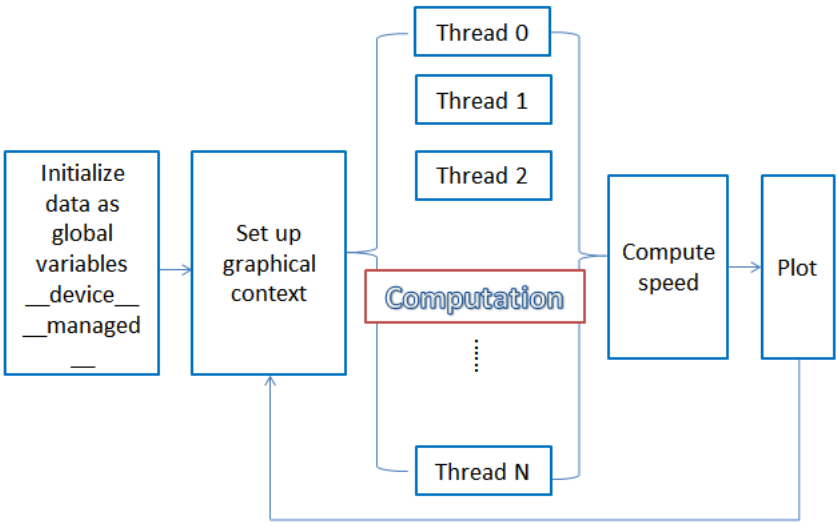
In the main function, obtain thread number from command line. The computation part shows as follows.

```
/* Start calculation */
auto beginCal = high_resolution_clock::now();
finished = true;

// Computation
calculate<<<1, thread_num>>>();

// Wait for computation to finish
cudaDeviceSynchronize();

first_buffer = !first_buffer;
auto endCal = high_resolution_clock::now();
/* Finish calculation */
```

We use the kernel function *calculate* to distribute tasks to GPU. Remember to synchronize the threads after calling the kernel functions since the control returns to CPU immediately and it should wait for all computation tasks to finish.

Here I only use one 1-D block, making the thread number decisive. The reason for this is to simplify the code logic. If the grid/block number is varied, there are multiple methods to determine the thread dimensional arrangement. Also, the data synchronization between blocks is difficult and time-consuming, which causes unnecessary performance cost. Only one block is sufficient to realize parallel programming.

The CUDA program flow:




- Bonus

The bonus part combines both MPI and OpenMP. We distribute the tasks to different processes. In each sub-task, we can also make use of the openMP #pragma to further distribute the workload. It is a multi-process + multi-thread programming.

The main function part is quite similar with that of MPI version. The only difference is that a thread number is entered by the user to implement openMP. The thread number is then broadcast to all processes for sub-division.

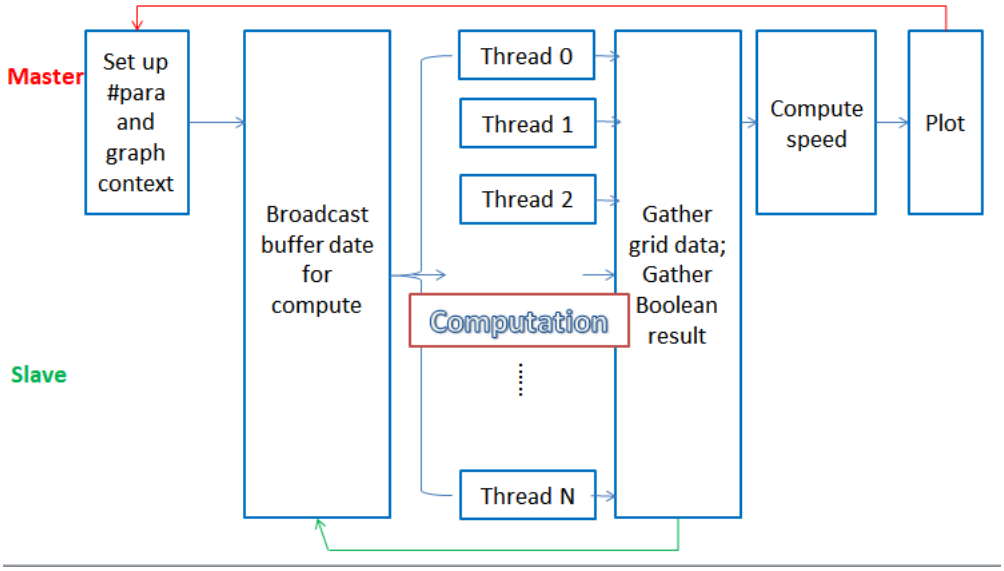
```
omp_set_num_threads(thread_num);
MPI_Bcast(&thread_num, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

In the hdist.hpp (computation part), the code is then quite similar with that of openMP version, although we just need to add one extra **#pragma**.

Use start and end indexes to divide the task (MPI). Then in the computation between the bodies, the #pragma is used in **internal** for loop (divide rows), since we already divide the columns and it is not a good idea to continue partitioning. No private variables need to be specified since all threads share the same memory, the updating is on the correct positions. In this way, each thread is assigned to a block of area in the square room. Block division!

```
case Algorithm::Jacobi:
    for (i = start; i < end; ++i) {
        #pragma omp parallel for
        for (j = 0; j < state.room_size; ++j) {
            auto result = update_single(i, j, grid, state);
            stabilized &= result.stable;
            grid[{alt, i, j}] = result.temp;
        }
    }
    grid.switch_buffer();
    break;
```

The program flow:




- Some details

Only the computation part makes sense when calculating the speed, since plotting costs greatly by a single thread.

The program runs repeatedly to average the speed and obtain more accurate results.

The method I use is static allocation, partitioning the room by their *i-index* into column groups, which balances the workload to some degrees.

Represent the 2-D room into a 1-D vector requires multiple of the room length but I forget to multiply, which cost me a great amount of time to debug……

Add **set(CMAKE_CXX_FLAGS "-fsanitize=thread")** in the Cmakefile to check data race.

# 3 Execution

The execution for my code performs on the remote cluster. There are 8 internal nodes each with one **NVIDIA 2080Ti GPU card** with 32 cores together. The maximum CPU time limit is 640 (e.g. 4 nodes, 128 cores for 5 minutes).

Environment: Linux server equipped with multi-thread library and NVIDIA GPU.

***Way to compile and execute (MPI, Pthread, and OpenMP):***

1. Unzip 118010141.zip
2. cd csc4005-assignment-4 & csc4005-imgui-xxx (mpi, pthread, openmp)
3. mkdir build && cd build
4. cmake ..
5. make

```
cd /path/to/project
mkdir build && cd build
cmake .. -DCMAKE_BUILD_TYPE=Debug # please modify this to `Release` if you
want to benchmark your program
cmake --build . -j4
```

6. *MPI:* **mpirun -np 4 ./csc4005_imgui** [Can specify the process number here]

   *Pthread/OpenMP:* **./csc4005_imgui 4** [Can specify the thread number here]

7. Adjust the parameters (in the *hdist.hpp* in code) to obtain different effects.

8. Adjust *block_size* to change the size of the plot.

9. Check the duration and speed information in the bash.

10. **Ctrl+C** to **forcedly** end the program.

***Way to compile and execute (CUDA):***

1. cd csc4005-imgui-cuda
2. mkdir build && cd build
3. source scl_source enable devtoolset-10
4. CC=gcc CXX=g++ cmake ..
5. make
6. (Add *srun* in the server) **./csc4005_imgui 4**

   [Can specify the thread number here]
7. Adjust the parameters (in the *hdist.hpp* in code) to obtain different effects.
8. Adjust *block_size* to change the size of the plot.
9. Check the duration and speed information in the bash.
10. **Close** the GUI window to stop the program

Some notice:

For **graphical exception**:

Use **ssh –Y** to register.

In /pvfsmnt/118010141 directory:

Copy XAuthority file:

```
cp ~/.Xauthority /pvfsmnt/$(whoami)
```

Set XAuthority env:

```
export XAUTHORITY=/pvfsmnt/$(whoami)/.Xauthority
```

For most of the experiment, I use ***salloc*** to run interactively:

**Salloc –N1 –n32 –t10**

I also use ***sbatch*** to submit the task for some experimental group. A sample script is shown as follows:

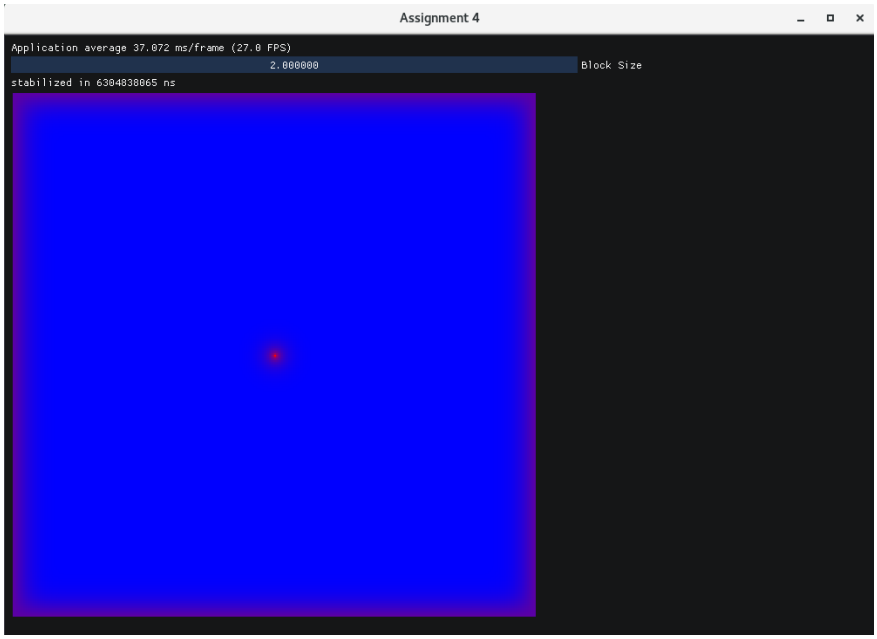
```
#!/bin/bash
#SBATCH -o cuda64.out
#SBATCH --account=csc4005
#SBATCH --partition=debug
#SBATCH --qos=normal
#SBATCH --nodes=4
#SBATCH --ntasks=128
#SBATCH --time=03:00

srun /pvfsmnt/118010141/csc4005_imgui 128
```

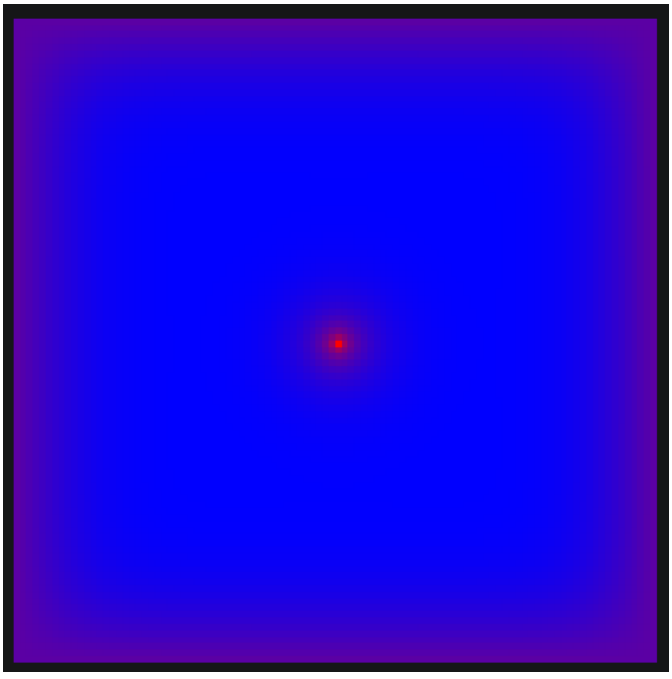
Again, the program should exit after a certain time limit defined by the *sbatch* script or forcedly ended by the user [ctrl + C] since it is running repeatedly.

# 4 Result & Analysis

- GUI Result

The GUI window is shown as follows. For performance and code logic consideration, no draggers for parameter are provided here. (The network communication speed of GUI is amazingly slow between server and local host). The following shows room size 300*300 and 100*100, stabilized state.



**Two videos** are attached in the homework folder for animation display. The first video runs under the MPI configuration with 4 processes. The room size is 100 * 100 and block_size is 5 and tolerance is 0.01. As time goes by, the temperature of points will increase. The closer the point to the fire source or the wall, the higher the value is. (**Here the speed is not stable since the room size is small, when the room size is default value 300, the speed is stable, but the coloring effect is not obvious, so it is for experiment setting but not animation display**)

The second video is about openMP configuration with 2 threads and room size is 50 * 50 and tolerance is 0.05. Block size will change. Modification to different parameters will bring different effects, which will be illustrated below.

- Bash Result

In the bash, the computing time (in nanoseconds) is shown. The speed is calculated and output then (iterations/second). The output is continuous.

```
^Cbash-4.2$ mpirun -np 5 ./asg4_mpi
[WARN] cannot get screen dpi, auto-scaling disabled
libunwind: __unw_add_dynamic_fde: bad fde: FDE is really a CIE
libunwind: __unw_add_dynamic_fde: bad fde: FDE is really a CIE
libunwind: __unw_add_dynamic_fde: bad fde: FDE is really a CIE
10 elapse with 17687314 nanoseconds
speed: 565.377 iterations per second
10 elapse with 17526670 nanoseconds
speed: 570.559 iterations per second
10 elapse with 17442718 nanoseconds
speed: 573.305 iterations per second
10 elapse with 17037664 nanoseconds
speed: 586.935 iterations per second
10 elapse with 17220920 nanoseconds
speed: 580.689 iterations per second
10 elapse with 17036269 nanoseconds
speed: 586.983 iterations per second
10 elapse with 16899418 nanoseconds
speed: 591.736 iterations per second
10 elapse with 17106827 nanoseconds
speed: 584.562 iterations per second
10 elapse with 16737237 nanoseconds
```

MPI

```
bash-4.2$ srun ./csc4005_imgui 2
[WARN] cannot get screen dpi, auto-scaling disabled
10 elapse with 144965100 nanoseconds
speed: 68.9821 iterations per second
10 elapse with 144572422 nanoseconds
speed: 69.1695 iterations per second
10 elapse with 144445360 nanoseconds
speed: 69.2303 iterations per second
10 elapse with 144464751 nanoseconds
speed: 69.221 iterations per second
10 elapse with 144437275 nanoseconds
speed: 69.2342 iterations per second
10 elapse with 144551398 nanoseconds
speed: 69.1795 iterations per second
10 elapse with 144500250 nanoseconds
speed: 69.204 iterations per second
10 elapse with 144509507 nanoseconds
speed: 69.1996 iterations per second
10 elapse with 144452024 nanoseconds
speed: 69.2271 iterations per second
10 elapse with 144458262 nanoseconds
speed: 69.2241 iterations per second
10 elapse with 144399926 nanoseconds
speed: 69.2521 iterations per second
10 elapse with 144408696 nanoseconds
speed: 69.2479 iterations per second
10 elapse with 144459180 nanoseconds
speed: 69.2237 iterations per second
10 elapse with 144294766 nanoseconds
speed: 69.3026 iterations per second
10 elapse with 144375015 nanoseconds
speed: 69.2641 iterations per second
10 elapse with 144446118 nanoseconds
speed: 69.23 iterations per second
10 elapse with 144431114 nanoseconds
speed: 69.2372 iterations per second
stabilized in 93688029308 ns and 170 iterations
```

```
bash-4.2$ srun ./csc4005_imgui 128
[WARN] cannot get screen dpi, auto-scaling disabled
[WARN] cannot get screen dpi, auto-scaling disabled
[WARN] cannot get screen dpi, auto-scaling disabled
[WARN] cannot get screen dpi, auto-scaling disabled
10 elapse with 49004321 nanoseconds
speed: 204.064 iterations per second
10 elapse with 48629631 nanoseconds
speed: 205.636 iterations per second
10 elapse with 48627643 nanoseconds
speed: 205.644 iterations per second
10 elapse with 48578185 nanoseconds
speed: 205.854 iterations per second
```

CUDA

```
[csc4005@localhost csc4005-imgui-openmp]$ ./csc4005_imgui 128
[WARN] cannot get screen dpi, auto-scaling disabled
10 elapse with 2737747183 nanoseconds
speed: 3.65264 iterations per second
10 elapse with 2549557795 nanoseconds
speed: 3.92225 iterations per second
10 elapse with 2918957494 nanoseconds
speed: 3.42588 iterations per second
10 elapse with 2280293359 nanoseconds
speed: 4.3854 iterations per second
```

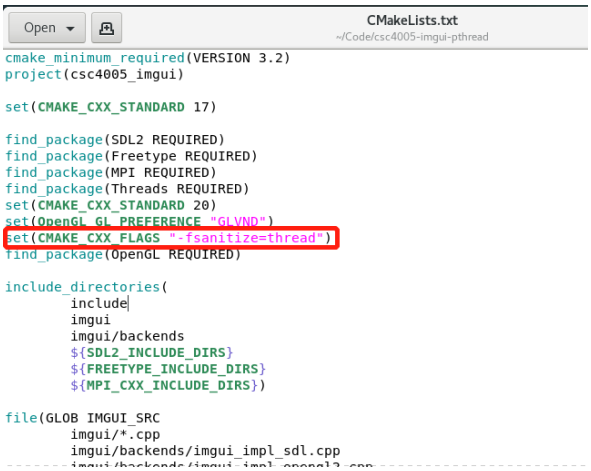
openMP

```
[csc4005@localhost csc4005-imgui-mpi+openmp]$ mpirun -np 2 ./csc4005_imgui
Enter thread number in MPI sub-task: 2
[WARN] cannot get screen dpi, auto-scaling disabled
10 elapse with 203793495 nanoseconds
speed: 49.0693 iterations per second
10 elapse with 259120846 nanoseconds
speed: 38.592 iterations per second
10 elapse with 247394347 nanoseconds
speed: 40.4213 iterations per second
10 elapse with 168795567 nanoseconds
speed: 59.2433 iterations per second
10 elapse with 230730294 nanoseconds
speed: 43.3406 iterations per second
10 elapse with 236001260 nanoseconds
speed: 42.3727 iterations per second
```

MPI + openMP

The speed may be unstable due to the process communication or thread synchronization or network problem.

Add a sentence to check data race. **No data race during computation appears in my program**.




There may be some strange warning by the checker at the very beginning of the code for openmp version (data race in one byte of mutex??), but **it should not be the data race during the computation** (should be size 8 bytes), I guarantee no data race happens **after successfully launch** the computation part and enter iteration.

- Robustness

The program is able to deal with the case with different process/thread numbers (1-N), different room size, source position, tolerance, temperature, and so on. The program can handle the case where room size length is smaller than process/thread number.

Before the analysis, let me state the default setting for the parameters.

**[room size = 300, block_size = 2, source_x = 150, source_y = 150 (middle), Source_temp = 100, border_temp=36, tolerance = 0.1, iteration=10]**

- Relationship between speed & process(thread) number

To explore the relationship between running speed in iterations per second (y-axis) and the process (thread) number (x-axis), the following experiments are presented.

Using the control-variable method, the room size, source position, temperature, tolerance and iteration should be fixed among different process/thread numbers. In the experiment group, I only set 1 group of room size (300), focusing more on the comparison between different implementation (MPI, Pthread, openMP, CUDA). This is because the server is too crowded and resources are scarce so it is hard to run many groups of data smoothly.

However, **I have analyzed the relationship between speed and room size (output size) separately in the below part**.

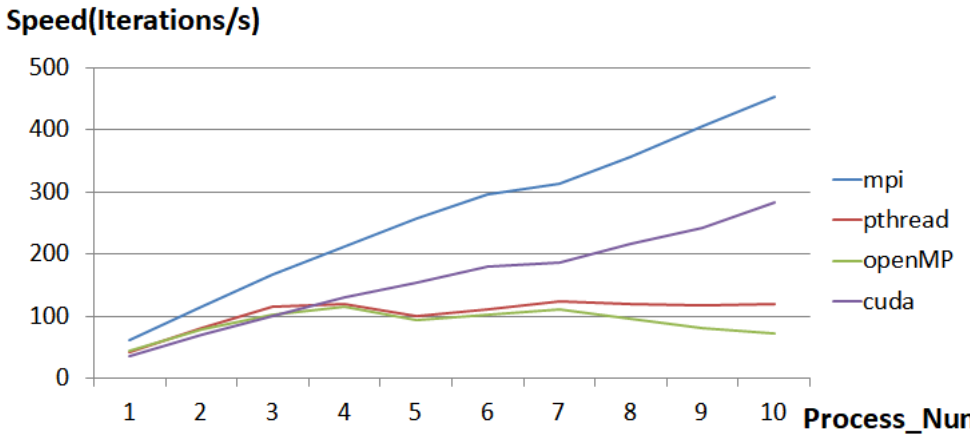
In each group, process/thread number from **1** (sequential) to **10** (parallel) will be tested (if the number is larger, the graphical window responses pretty slow, so 10 is sufficient to show some insights), whose speed (iterations / duration time) will be recorded and graphed in a line chart to show the comparison and analyze the tendency. ***[More than 33 processes/threads setting will be analyzed in the later part separately]***

The detailed data and the line chart are shown as follows.

The unit for the value is **iterations/s**, which will not be explained again later.

| Process/ Thread Number | mpi | pthread | openMP | cuda |
|---|---|---|---|---|
| 1 | 60.8 | 42.6 | 44.2 | 35.32 |
| 2 | 116.1 | 81.5 | 77.9 | 69 |
| 3 | 166.3 | 114.6 | 102.9 | 101.1 |
| 4 | 211.6 | 120.2 | 114.2 | 130 |
| 5 | 257.9 | 101.1 | 93.5 | 155 |
| 6 | 296.9 | 110.1 | 102.6 | 178.9 |
| 7 | 314.3 | 124.6 | 110.8 | 186.9 |
| 8 | 355.7 | 119 | 95.2 | 216.8 |
| 9 | 406.7 | 117.8 | 79.7 | 242.7 |
| 10 | 453.1 | 120.6 | 71.1 | 283 |

Relationship between speed & process number (1-10) MPI, Pthread, openMP, CUDA




As can be seen from the raw data and the graph, the program runs the most slowly sequentially in all of the four versions. The performance is quite close in sequential scenario, increasing as process/thread number goes up, and then separate into different path. MPI performs the best among the four, and then it is CUDA version. The internal implementation for pthread and openMP is the same so they have pretty similar performance! But their speed does not increase but decrease when thread number continues to add.

Notice that pthread and openMP versions experience a bottleneck from 4 to 5. When the thread number is larger than 4, no more speedup is gained in both pthread and openMP versions! This may be partly because of the ***thread initialization cost***, but the major cause should be the **CPU core limit**. The threads make a false appearance that the value can reach infinite; however, only one thread can run simultaneously in one core, corresponding to one processor. When the thread number is greater than the processor number, the operating system will run the threads **alternately**, forming a fake phenomenon of multiple threads, with the performance bottleneck in fact **the number of the cores**. I run the Pthread and openMP in my **virtual machine** whose core number is **4**. As a result, these two versions have a bottleneck near 4. Pthread performs slightly better than openMP. In openMP the thread initialization is implicit and requires no care from programmer. Unfortunately, the actual speed in bash for Pthread and openMP version is not that perfect. The results also fluctuate a lot (in a certain range).

For the MPI and CUDA versions, the speed increases linearly as process number increases, forming a **nearly perfect straight line**! I run the experiment on the remote server, which supports multi-core computation. The slope is fixed, meaning that the computation speed is nearly the same among different processes. The result shows the advantage of my approach for distribution——statically by columns. The balanced workload renders each process similar time to finish the task and return the results back for plotting. There is little interference issue like process synchronization or data

transfer events when the process number is relatively small (1-10). In MPI version, the communication does not cost much to change the linear shape, and initialization of processes does not count! In CUDA version, the modification is directly to the global variable __device__ __managed__, even smaller extra cost besides computation. The two results are reasonable.

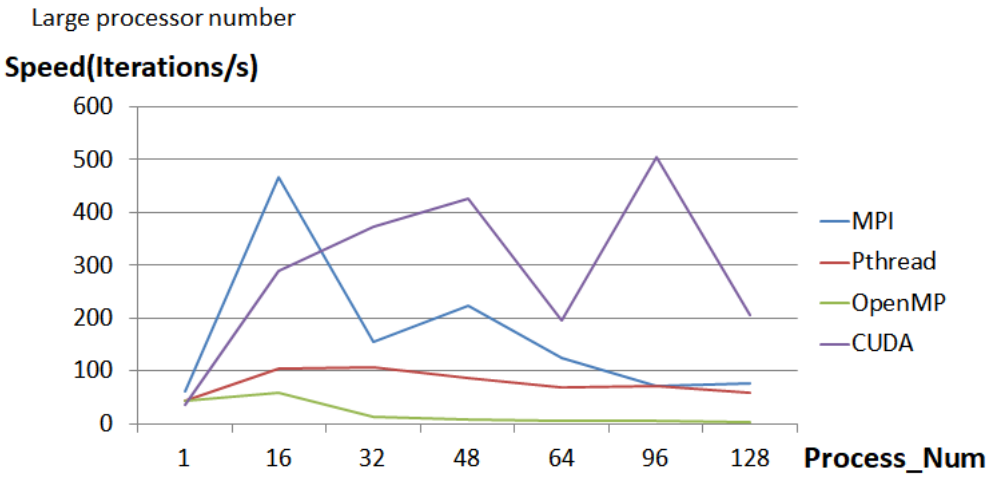
In conclusion, the MPI and CUDA version performance is pretty stable and fits the expectation when the process number is in range 1 to 10. The Pthread and openMP versions meet a bottleneck around thread number 4, increasing from 1 to 4 and decreases generally from 5 to 10. The performance rank is MPI-CUDA-Pthread-OpenMP in 1-10 process/thread numbers.

**## Large process number**

I made another experiment, setting the number of processes sequential and the multiple of 16. Because every node has 32 cores, I would like to explore the efficacy when the cores are distributed in different (2-4) GPU nodes for MPI & CUDA version. To make a comparison, the Pthread and openMP versions data are also shown.

| | MPI | Pthread | OpenMP | CUDA |
|---|---|---|---|---|
| 1 | 60.8 | 42.6 | 44.2 | 35.32 |
| 16 | 465.705 | 103.5 | 57.9 | 290.3 |
| 32 | 154.4 | 108.1 | 12.8 | 373.4 |
| 48 | 223.4 | 86.1 | 6.74 | 427 |
| 64 | 125.8 | 69.5 | 5.81 | 196.4 |
| 96 | 72 | 71.4 | 4.33 | 503.9 |
| 128 | 76.4 | 59.2 | 3.65 | 205.6 |

Relationship between speed & process number (1-128)




From the raw data and the graph, for MPI, the running speed increases greatly from 1 process to 16 processes but falls greatly to a certain level at process 32 and increases slightly at 48 and then becomes worse with even larger process number. The speedup is no longer perfectly linear when the core number is relatively huge. Because the cost for process communication and synchronization increases as the number of processes adds up this time. The main cost lies in the **data broadcast and gathering**. The room size is small, and each process obtains only several columns during one elapse. The pure computation time is pretty small and outweighed by the communication time. Besides, the cost for process communication between different nodes is far greater than that in the same node (32 cores). In the same GPU, the cost is ignorable (proved by my previous experiment 1-10), but in the different GPU, the cost cannot be neglected! As a result, structure of the cluster should also be taken into account when analyzing. There are 32 cores in a node, and every one node addition will reduce the speedup ration (the slope in the graph).

Also, for the CUDA version, the line fluctuates greatly, meaning that CUAD version also reaches its bottleneck. But the situation is slightly different. The CUDA version keeps increasing from 1-16-32-48 and then falls in 64 and increases in 96 and decreases at 128 at last. There are plenty of threads in a single block, up to tens or even hundreds. The communication between threads can be ignored in the same block, but extraordinary between different block or the GPU. The performance is highly related to the internal structure of NVIDIA GPU.

For Pthread versions, conforming to my previous analysis, the performance is stable \after certain thread number. But for the openMP, the performance decreases generally and reaches close to 0 but not maintain at a certain level.

One of the possible reasons (except CPU core bottleneck) to explain the bottleneck and the instability is the cost for thread creation and synchronization. In the main thread, the other threads are created one by one, causing different start time for computation. One thread can only start after the previous thread is successfully launched. We should read source code for pthread and openMP to gain more insights.

As a result, when the thread number is large, the time interval between the first thread and the last thread should be large.

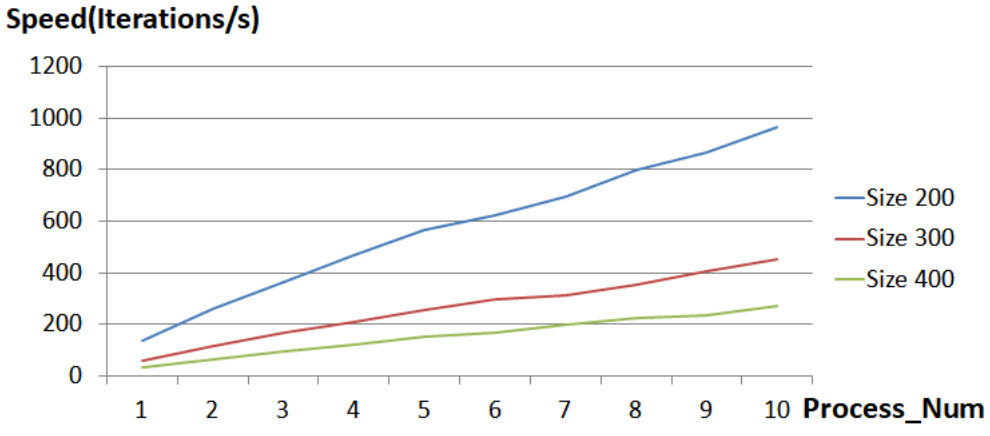
After computation, the function Pthread_join() is used to synchronize all the threads, causing a time delay. Assume the time for creating a thread is fixed, it is normal that when the thread number increases, although the computation time can be shortened, the time difference increases, causing a longer waiting time for thread synchronization and no improvement on the total performance due to the elimination.

- Relationship between speed & room size (output size)

The following experiments are conducted in MPI configuration since it is relatively stable. The data is crazy for Pthread and openMP, varying greatly. Also, running CUDA is a little bit troublesome.

Similar to the previous experiment, to explore the relationship between the speed and room size (output size), the number of processes/threads (we observe vertically) and other parameters should be fixed. Here I choose process number 1-10, too. The room size varies from 200, 300 to 400, with 100 additions each time. The results are shown as follows.

| Process Number | Size 200 | Size 300 | Size 400 |
|---|---|---|---|
| 1 | 137.4 | 60.8 | 34.1 |
| 2 | 259.9 | 116.1 | 65.3 |
| 3 | 366.9 | 166.3 | 93.6 |
| 4 | 467.1 | 211.6 | 120.7 |
| 5 | 565.4 | 257.9 | 152.1 |
| 6 | 625.8 | 296.9 | 167.9 |
| 7 | 694.7 | 314.3 | 199 |
| 8 | 797.6 | 355.7 | 224.6 |
| 9 | 864.9 | 406.7 | 233.9 |
| 10 | 964.2 | 453.1 | 272.1 |



As can be seen from the raw data and the graph, the speed decreases obviously when room size increases. For MPI version, the tendency is true, fitting our previous conclusion that running time increases with the problem size N. The larger the room size, the heavier the workload is, thus requiring more computational resources. In each process or thread, it is allocated with more rows to calculate (same columns), and the communication and synchronization efforts also increase. The time complexity is $\mathbf{O(N^2)}$, and the nearly 4/9, 9/16 ratio proves its correctness (after computation for some groups).

The perfect straight line illustrates that MPI performs stably and the problem size will make minimal influences on its fixed speed property among processes or process synchronization/communication process.

- Relationship between speed and other parameters

For other parameters, actually they do not make much sense in exploring their relationship with speed. Block size is only related to plotting. Position of source affects the temperature distribution. Temperature values are related to color. Tolerance is related to the total time to stabilize and converge. As a result, exploring the relationship between speed and process/thread number, speed and room size are sufficient to show some interesting conclusions.

- Performance of MPI + OpenMP version

The following shows some simple data of bonus part:

| Thread / Process (iteration/s) | 1 process | 2 processes |
|---|---|---|
| 1 thread | 43.3 | 79.2 |
| 2 threads | 39.4 | 47.5 |

As can be seen from the raw data, MPI does bring performance improvement, checking horizontally. However, openMP reduces the speed…… Maybe my computer architecture does not support the MPI + OpenMP multi-process + multi-thread implementation. The openMP needs to create threads in each process, but each process corresponds to one processor and no more processor to execute the task. The experiment result violates theoretical results.

# 5 Summary

Here I will summarize what I have learned when writing assignment 4.

- Understanding of parallel computing. Understanding of share memory multi-process/multi-thread programming.
- Special attention on writing parallel programs (process synchronization, process communication, thread creation and join, cost, **data race**)
- MPI & Pthread & OpenMP & CUDA library usage
- Coding ability improvement
- Analysis ability increases
- The ability to make use of searching engine
- The ability to utilize “wheel”.
- Patience……**The queue is sometimes too crowded!!!**
- Cluster experience. Queuing to obtain the resources!
- By the way, tutorials are quite practical for writing the project and TA & USTF are warm-hearted in answering the questions in the wechat Group, Kudos!

That’s all.