# Project 4 Heat Simulation

CSC4005 Distributed and Parallel Programing

Ryan Christopher - 119010507

School of Data Science

The Chinese University of Hong Kong, Shenzhen

## Project Introduction

Presently, a simulation can be done using a variety of computer algorithm types. However, the expansion of the simulation's scope could make its execution take an absurdly long time. Therefore, in order to shorten the simulation time, we may need to boost processing power. However, as the speedup only increases by a factor of two, increasing the computing power of a computer may not be enough. Thus, parallelizing the simulation approach is a far superior option for resolving this problem. The behaviours of a sequential algorithm in comparison to certain parallel algorithms, including Message Passing Interface (MPI), POSIX Thread (Pthread), Open Multi-Processing (OpenMP), and CUDA, will therefore be examined in this study. The kernel version used is **3.10.0-1160.76.1.el7.x86_64.**

## Basic Understandings

The simulation of heat distribution is the issue that we'll use in this experiment. In general, the goal of this simulation is to demonstrate how heat energy moves from areas of higher energy to areas of lower energy.

The following formula in Figure 1 can be used to calculate the heat level of a pixel, which is how this experiment defines a pixel as the smallest area unit. The heat distribution of a room with four walls and a fireplace will also be simulated in this simulation. The heat level on certain walls and in the fireplace will remain constant during the simulation. They will therefore act as the simulation's heat source for other pixels.

## Sequential Programming

The program creates arrays to hold the heat value of each pixel in order to initialize the value for the sequential process. The values for the fireplace and the walls are then set by the program. After that, we'll have a loop where each iteration takes up one unit of time. The program will iterate through every pixel (aside from those associated with walls or fireplaces) for each iteration and calculate each pixel's heat value based on the neighbouring pixel's heat value (see the formula above). The outcome will be saved in a different variable so that the calculations of other pixels won't be thrown off by the current pixel's modified value. Additionally, the program will compare the new value to the previous one immediately after calculating each pixel. A checker variable will be set to indicate the need for the following iteration if the value exceeds a specific predetermined threshold. When all calculations are complete, the program will either move on to the next iteration or print the updated values using some graphical tools.

```
void maintain_fire(float *data, bool* fire_area) {
    // maintain the temperature of fire
    int len = size * size;
    for (int i = 0; i < len; i++){
        if (fire_area[i]) data[i] = fire_temp;
    }
}


void maintain_wall(float *data) {
    // TODO: maintain the temperature of the wall
    for (int i = 0; i< size; i++){
        data[i] = wall_temp;
    }
    for (int j = 0; j< size; j++){
        data[j * size] = wall_temp;
    }
    for (int k = 0; k< size; k++){
        data[(k*size) + (size -1)] = wall_temp;
    }
}
```

Figure 1. Benchmark algorithm to maintain wall temp and maintain fire

## MPI Programming

The master process in the MPI will initialize an array used to store the values for each overall pixel after the MPI initialization. The master process will then distribute the array to all accessible processes after initializing it (by setting the values for the walls and fireplace). According to the stripe-type data distribution, each process will receive a number of complete rows. We use this kind of data distribution because it makes it possible for us to share data more effectively during the simulation period. Then, each process will make local arrays to store its data once it has received data from all other processes. Each process will produce an array with two extra rows at the start and end of the array.

The last row from the previous process's data and the first row from the following process' data are kept in this extra space. With this design, we can set aside time to perform the data sharing between processes at the start of each iteration. Each process can start its calculations after the data sharing. Each pixel's output will also be kept in

another array. Each process must keep track of whether the iteration needs to continue or not, just like the sequential algorithm.

The checker variable from each process will be combined using reduce-max after all calculations are complete, and it will then be distributed to all processes. Each process is able to track the status of its execution in this way. Additionally, we might want to print the current iteration's illustration. In order to print the final product, all processes must collect their values at the time it occurs and send them to the master process.

Note: The code for MPI Programming is not completely finished. However, the theory and implementation are understandable

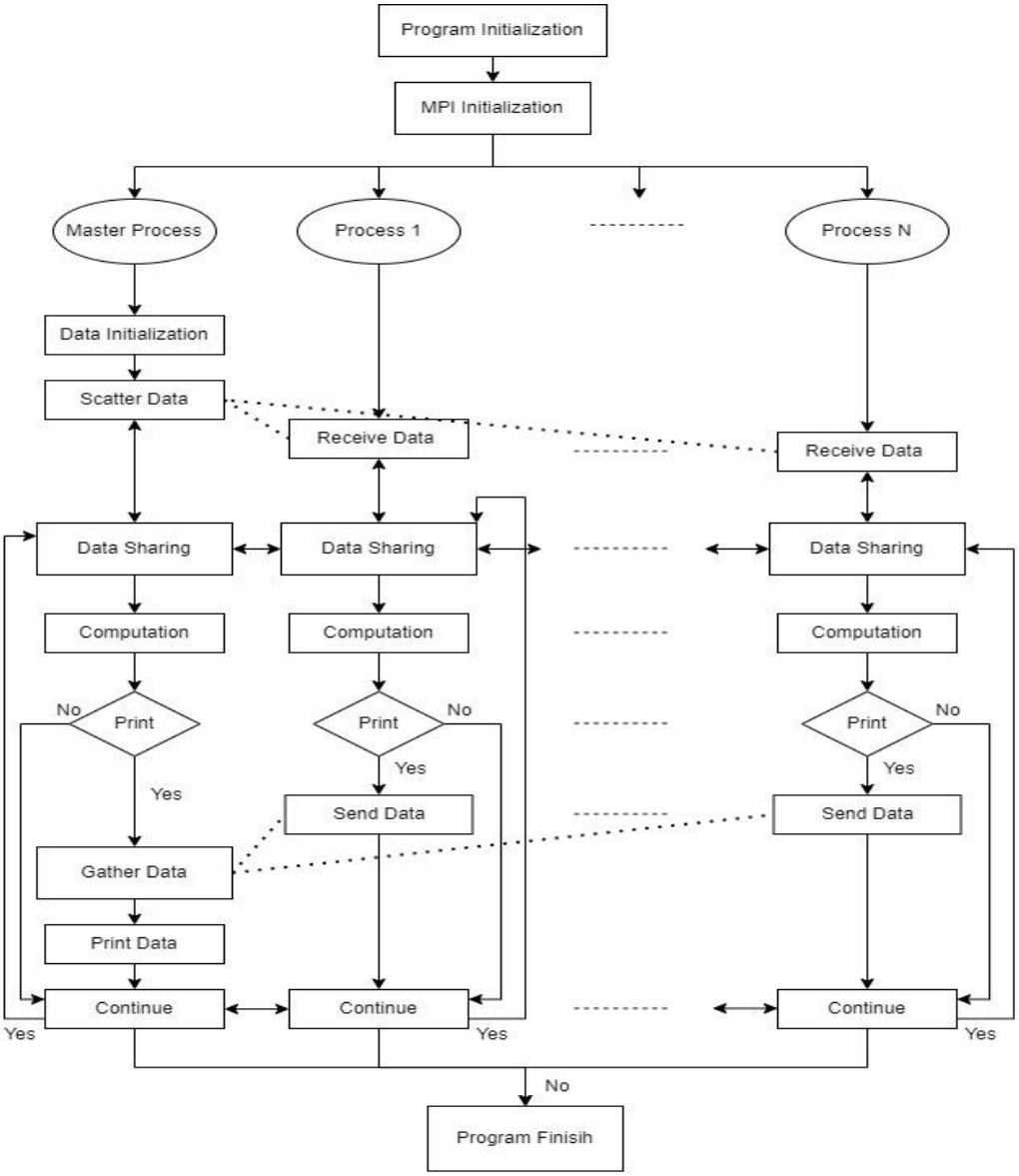


Figure 2. MPI Flow Chart Implementation

## Pthread Programming

We don't need to transfer data among the threads (thanks to shared memory), doing this simulation in Pthread is a little bit simpler. As a result, the software initially initializes the arrays with the values they would contain (walls and fireplace). Finally, threading is started by initializing the pthread after creating an array to record each thread's data distribution address. Because all threads share memory, simulation calculations for each one can begin very immediately after pthread setup. We also use a checker variable, which is shared by all threads, in the calculation of each pixel. Therefore, the program needs to add a barrier to synchronize all threads after all calculations are complete (the result is also stored in another array) before resuming the loop. During the synchronization phase, thread 0 will print the current result if the application needs to do so.

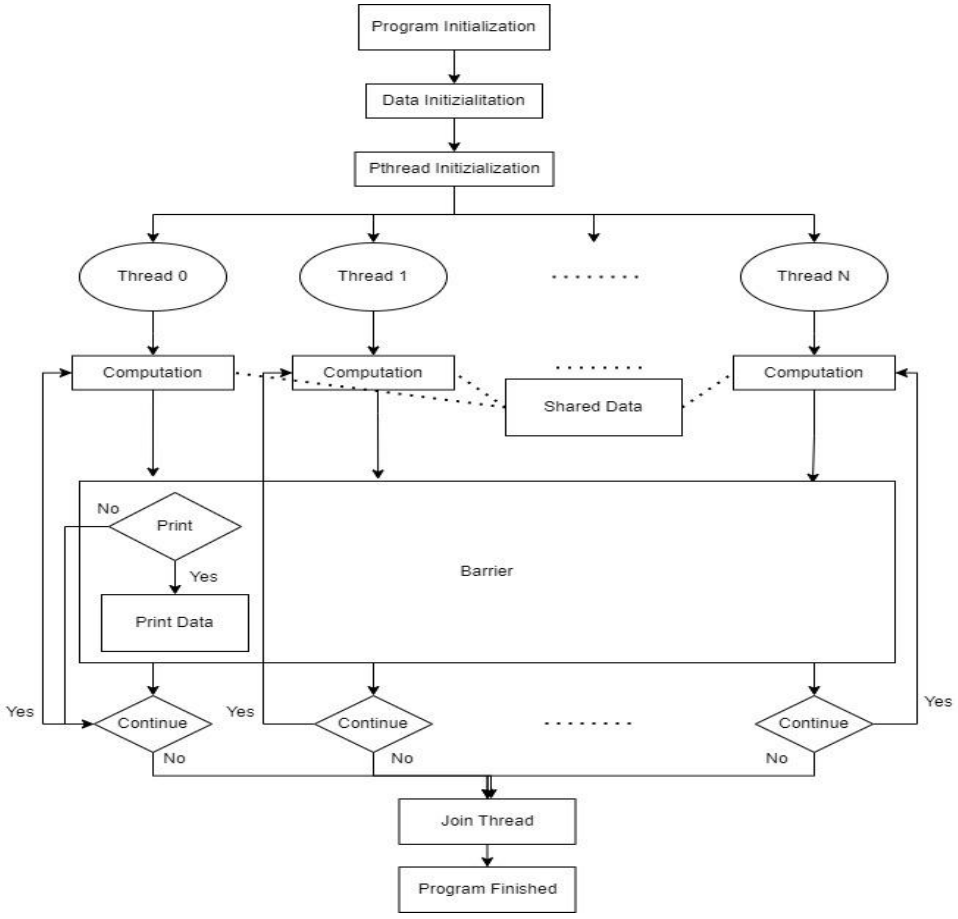


Figure 3. Pthread Flow Chart Implementation

## OpenMP Programming

The sequential algorithm has only been modified for the OpenMP implementation. In this application, the only modification is made before iterating through each pixel. The modification is the addition of a pragma that will parallelize (by employing threading) the repetition of pixels. In this case, two for loops are nested to actually implement the pixel iteration. Iterating through the vertical axis is done using the first loop, while iterating through the horizontal axis is done using the second loop. The first one, the vertical axis, is the one we parallelize. OpenMP and Pthread have quite distinct coding implementations, yet their internal workings are very similar. We want to parallelize the pixel iteration in both Pthread and OpenMP, which is why. The sole distinction is that in Pthread, threads are available at all times, whereas in OpenMP, threads are created and connected for each simulation cycle. As a result, Pthread might perform a little bit better.

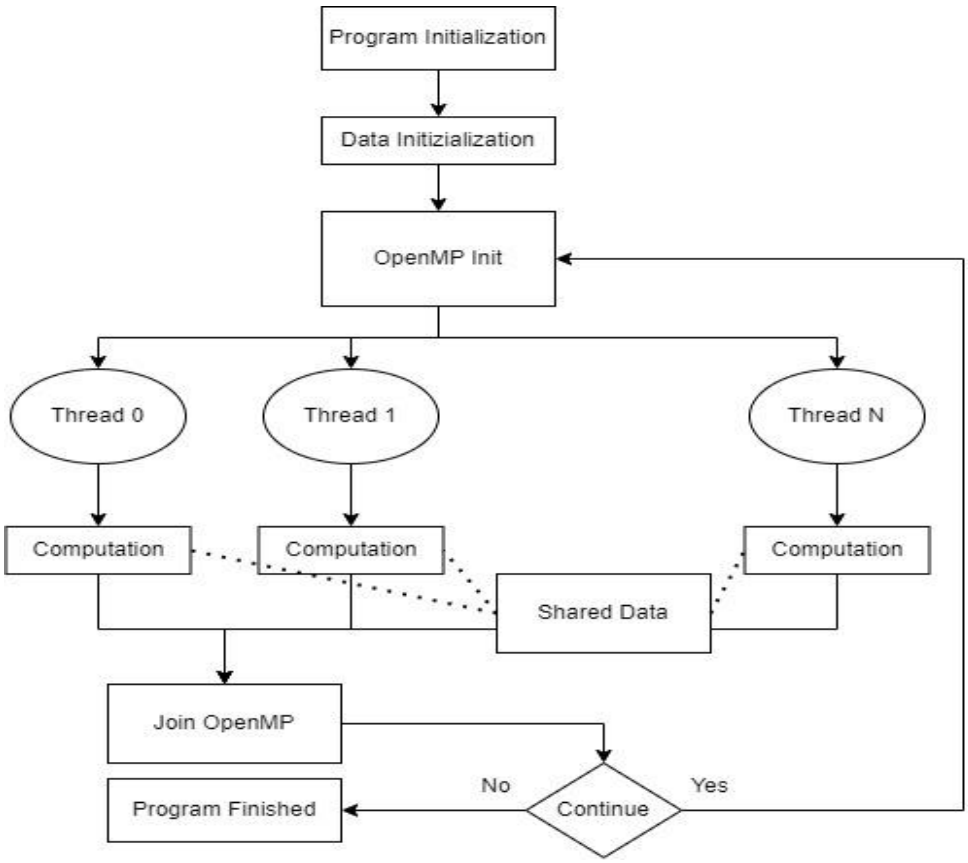


Figure 4. OpenMP Flow Chart Initialization

## Cuda Programming

The final parallel implementation makes advantage of the CUDA programming model from Nvidia. Comparing CUDA to earlier parallel implementations reveal how distinctive and diverse it is. The Graphical Processing Units (GPU) with their massive core count are used by CUDA. These cores are slower than standard CPU cores and can only handle rudimentary tasks. CUDA supports running several threads at once. A CUDA thread block is a collection of these threads. The block size in our software is set to 512. By using the distinct Blockidx and Threadidx, we may index each individual thread separately.

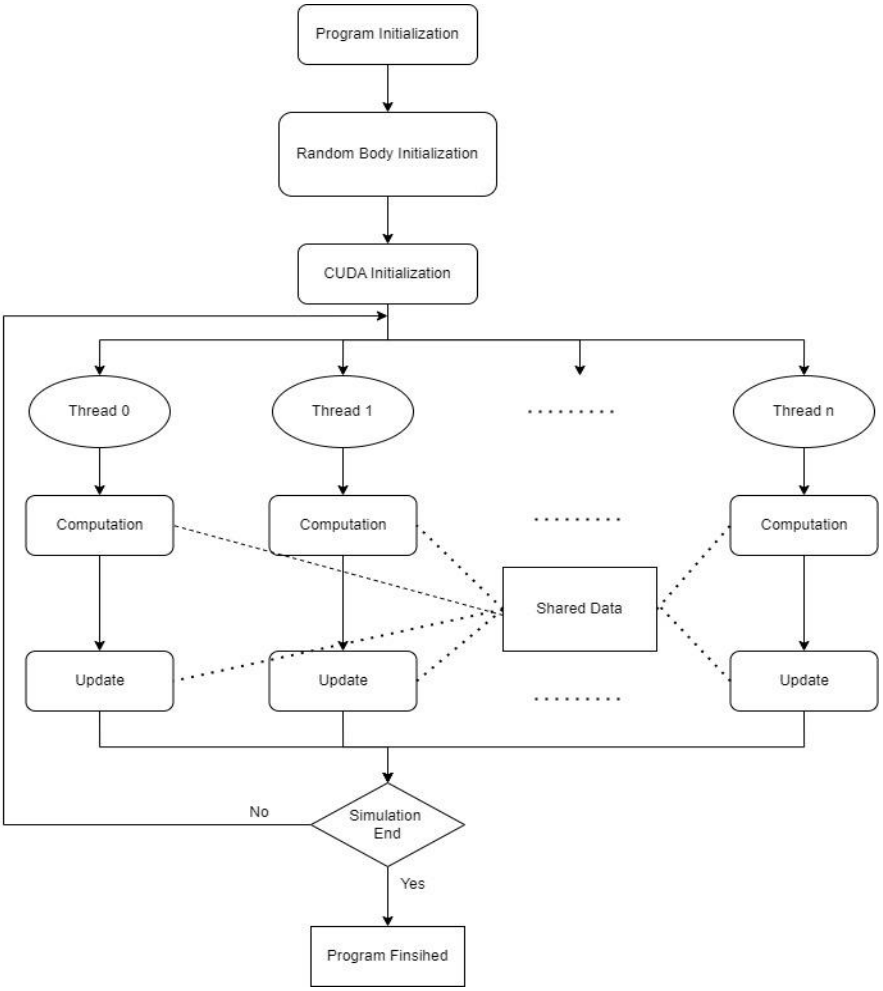


Figure 5. CUDA Flow Chart Implementation

# Result Experimentation and Analysis

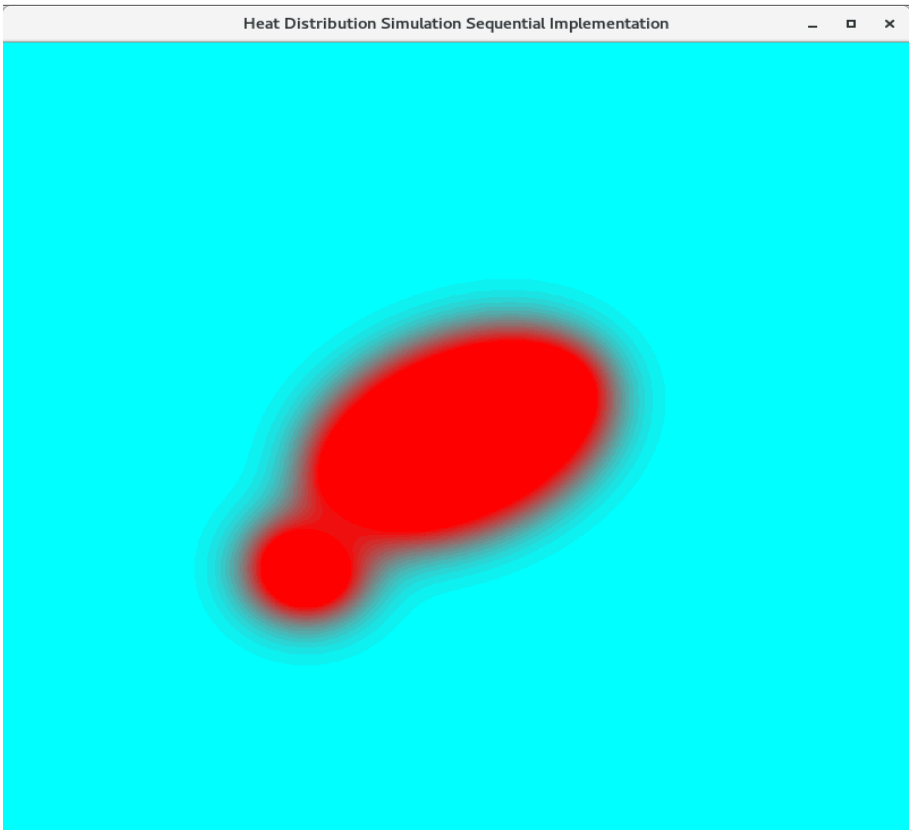


Figure 6. Heat Distribution Simulation Result

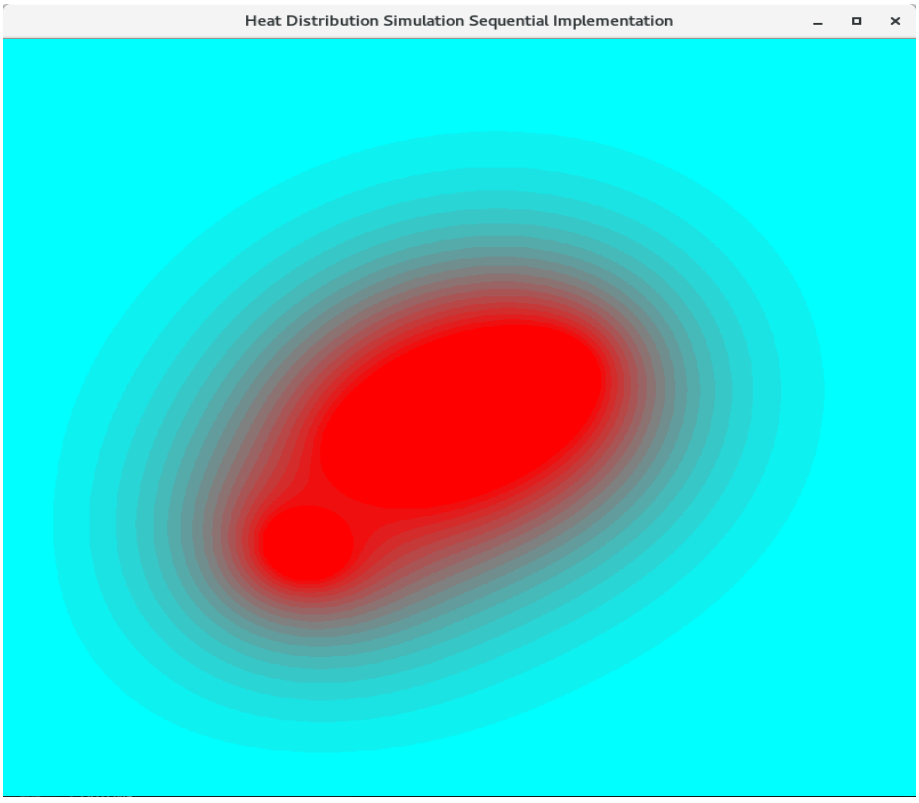


Figure 7. Heat Distribution Result after 5 to 10 minutes after figure 6

The image in Figure 6 displays the heat distribution simulation result. The blue section indicates the area with colder temperature while the red section indicates the area with hotter temperature. The initial state of the red section is small however as time elapsed it will distribute more toward the blue section as we can see in Figure 7.

the duration of all simulations using 100 × 100-pixel windows. We can see that almost all parallel techniques' lines exceed the sequential line in this situation. The simple fact that the Since parallel systems require so much time for communication, they offer more drawbacks than benefits. The line trends for Pthread and OpenMP exhibit a mildly positive gradient, indicating that having more threads results in longer communication times. Furthermore, further discussion is necessary regarding the notable improvements of OpenMP when used with four and five threads. When compared to the sequential employing one to four processes, the MPI line in the other image is pretty interesting. However, the running time dramatically increased when it employed more than four processes.

For the simulation with size 1000x1000, the time throughput for pthread stay is the bottom compared meaning that it runs the fastest compared to other parallel programming. Regarding multithreading, after four threads, it seems that implementations are still not getting better. The abrupt increase in the OpenMP line, however, appears to be significantly decreasing. This implies that the spike could eventually disappear as the windows grow in bigger.

## Project Summary

As a result, even if there are numerous algorithm designs for running simulation programs, we may still require more processing power. A more potent computer is typically difficult to construct, though. Therefore, this problem can be resolved via

parallel computing. In order to run the heat distribution simulation, this paper compared the performance of a sequential algorithm versus parallel techniques (such MPI, Pthread, and OpenMP). The outcome is astonishing because it is significantly better than the sequential implementation. However, each parallel computing system behaves differently from the others. Pthread produced the greatest results in this experiment, even though OpenMP was simpler to implement.