CSC4005 Distributed and Parallel Computing

Lab 4. Parallel algorithms of heat distribution simulation

Name: 刘浩霖
Student id: 115010192

Date: Nov 30th, 2018

The Chinese University of Hong Kong, Shenzhen

Introduction

Heat distribution simulation is a task that simulates the heat distribution over time. It is a challenging task because a second differential equation is required to be solved during the computation. However, there are a lot of approximations to reduce the amount of calculation, such as Jacobi iteration, and gradient descent method. In this experiment, the gradient descent method has a better approximation, which is chosen to be the algorithm to compute the temperature.

Because of the massive amount of computation during the simulation, effective parallel algorithms are required to speed up the computation. In this experiment, the parallel algorithms were implemented using MPI and Pthread individually.

Design

The designs of sequential algorithm, MPI implementation, and Pthread implementation to conduct heat distribution simulation are shown in Figure 1, Figure 2, and Figure 3 respectively.

Figure 1. Design of the sequential algorithm to conduct heat distribution simulation.

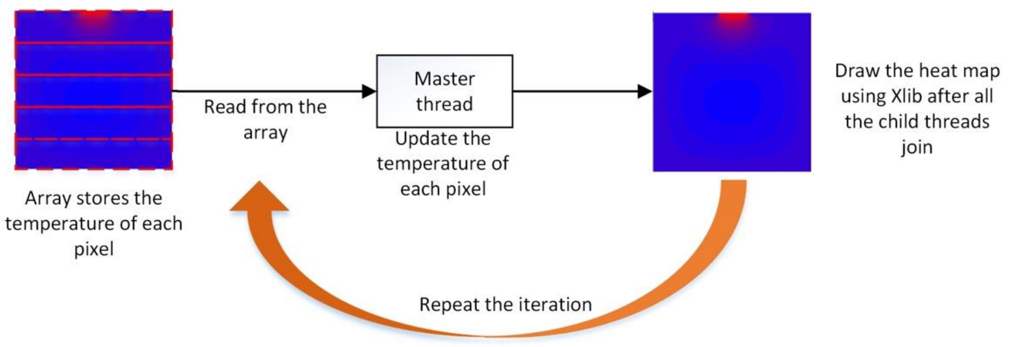

Read from the array
Master thread
Update the temperature of each pixel
Draw the heat map using Xlib after all the child threads join
Array stores the temperature of each pixel
Repeat the iteration

Figure 2. Design of the parallel algorithm to conduct heat distribution simulation using MPI.

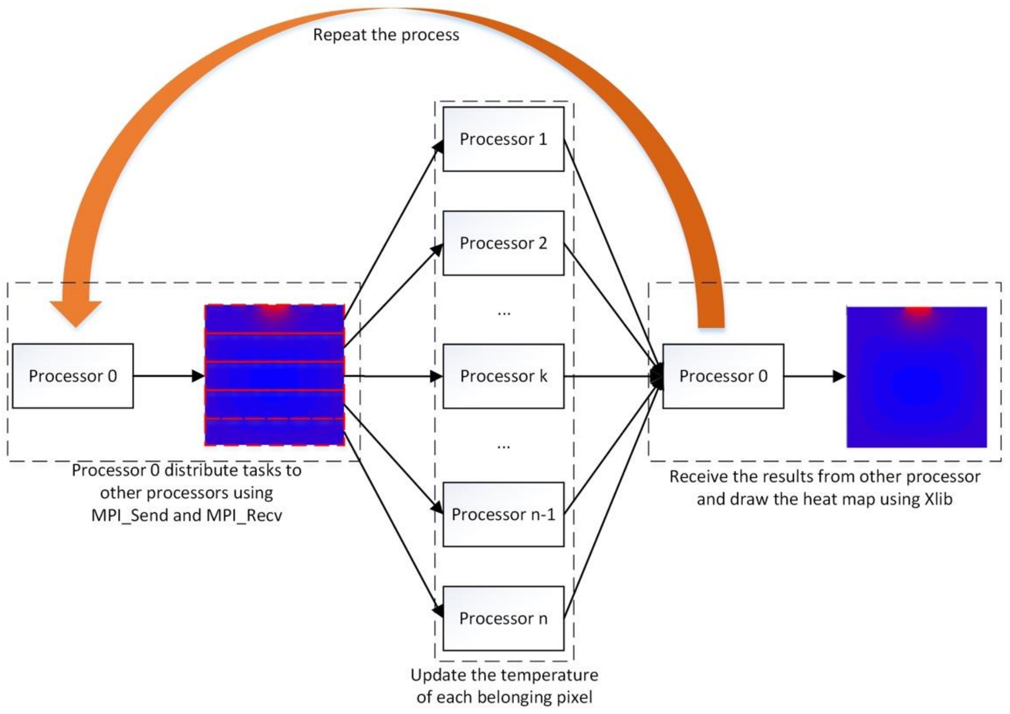
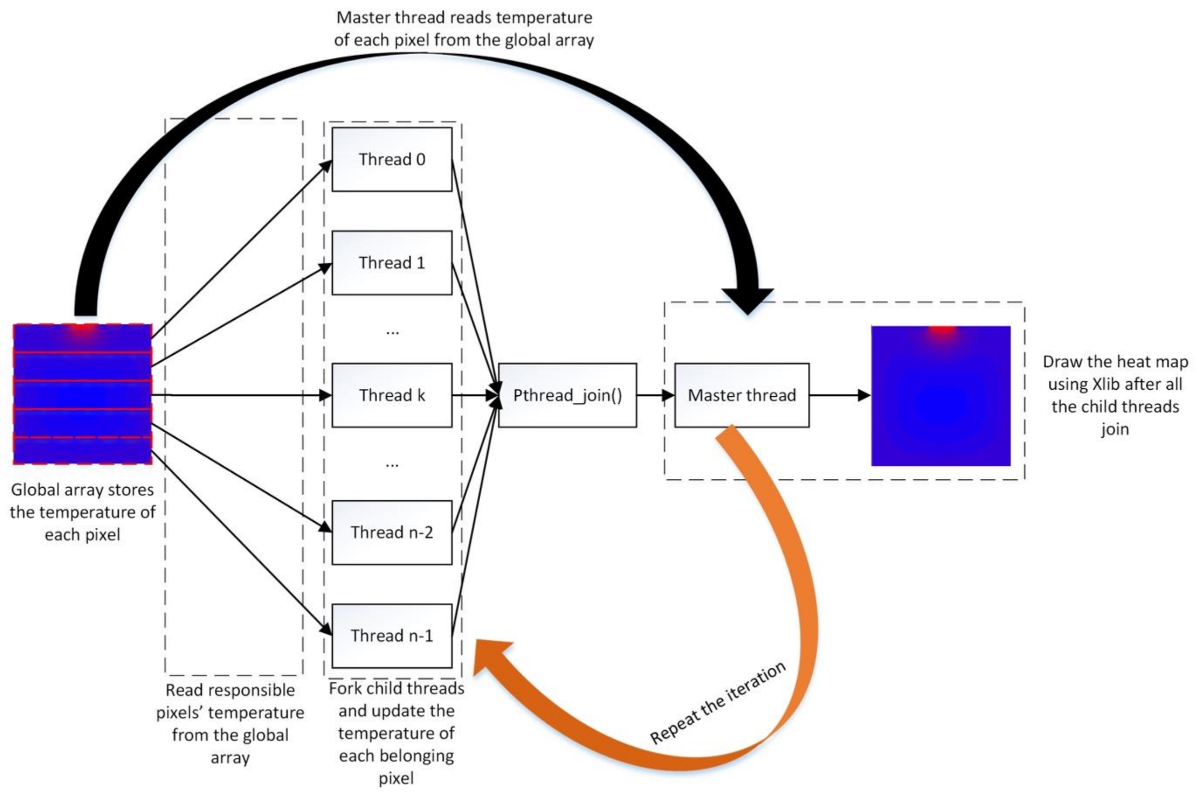

Repeat the process
Processor 1
Processor 2
...
Processor k
...
Processor n-1
Processor n
Processor 0
Processor 0
Processor 0 distribute tasks to other processors using MPI_Send and MPI_Recv
Update the temperature of each belonging pixel
Receive the results from other processor and draw the heat map using Xlib

Figure 3. Design of the parallel algorithm to conduct heat distribution simulation using Pthread.


Master thread reads temperature of each pixel from the global array
Thread 0
Thread 1
...
Thread k
...
Thread n-2
Thread n-1
Pthread_join()
Master thread
Draw the heat map using Xlib after all the child threads join
Global array stores the temperature of each pixel
Read responsible pixels' temperature from the global array
Fork child threads and update the temperature of each belonging pixel
Repeat the iteration

In Figure 1, the sequential algorithm to conduct the simulation is the simplest. [redacted] The gradient descent method is described below:

$$U(x, y, t+1) = \partial\{[U(x+1, y, t) - 2U(x, y, t) + U(x-1, y, t)] \\ + [U(x, y+1, t) - 2U(x, y, t) + U(x, y-1, t)]\}$$

In the above equation, $(x, y)$is the coordinate of the pixel, and $U(x, y, t)$ is the temperature of the pixel at (x, y) at time t. ∂ is the descending rate, in this experiment, $\partial = 0.01$. A smaller descending rate corresponds to more refined approximation, however, it would consume more computation time to reach equilibrium.

After the temperature is updated, the image is drawn; and after the image is drawn, the program would enter into the next iteration.

Figure 2 demonstrates the parallel algorithm implemented by MPI. [redacted]

[redacted] This is essential because this scheme allows the drawing and the computation ran in parallel, which saves a significant amount of time.

Figure 3 demonstrates the design of the parallel program using Pthread. [redacted] After the image is drawn, the master thread would enter into the next iteration and fork child processes to start the next update.

Result

The result of the sequential program, MPI program, and the Pthread program is shown in Figure 4, 5 and Figure 6 in which the image's size is 100*100, and red color indicates higher temperature, blue color indicates lower temperature; each color corresponds to a 5-degree temperature interval.

Figure 4. frames sampled from the resulting video of the sequential program

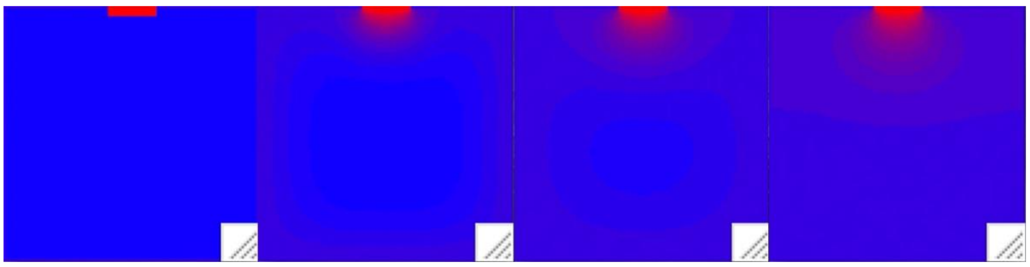


Figure 5. frames sampled from the resulting video of the MPI program

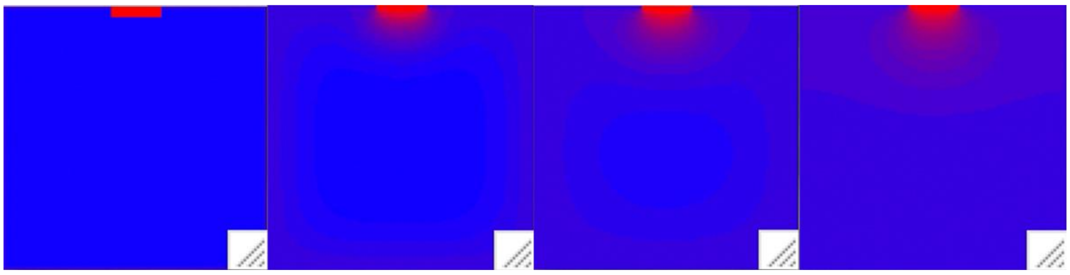
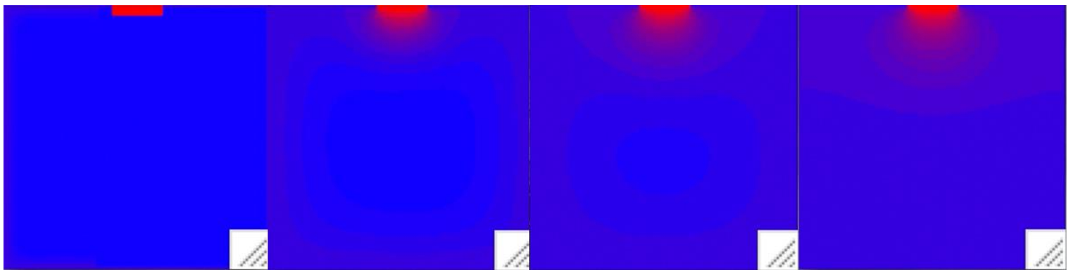


Figure 6. frames sampled from the resulting video of the Pthread program



Frames from the above three figures are consistent. The heat from the oven gradually spread in the room, however, the heat will not spread far and eventually reaches an equilibrium in which the temperature will be lower when it is further from the oven and the temperature will be higher when it is closer to the oven.

Performance analysis

The performance analysis is evaluated by the running time of 200 iterations. During the evaluation, the display of the result is switched off because the drawing process is extremely time-consuming on the cluster server. Otherwise, the running time difference between each configuration is trivial because the running time of every configuration is roughly the same. Therefore, the time consumed on the drawing is excluded from the evaluation. The performance analysis is conducted in two dimensions, varying in pixels' amount and the number of processors and threads.

The performance analysis of the sequential program is shown in Table 1. The performance analysis of the MPI program is shown in Figure 7, Figure 8 and Figure 9. The performance analysis of the Pthread program is shown in Figure 10, Figure 11, and Figure 12.

Table 1. Running time of the sequential program

PERFORMANCE ANALYSIS FOR THE SEQUENTIAL PROGRAM

Image size	Running time
100x100	0.0384s
500x500	0.9762s
1000x1000	3.8052s

Figure 7. Running time of the MPI program when the image size is 100x100

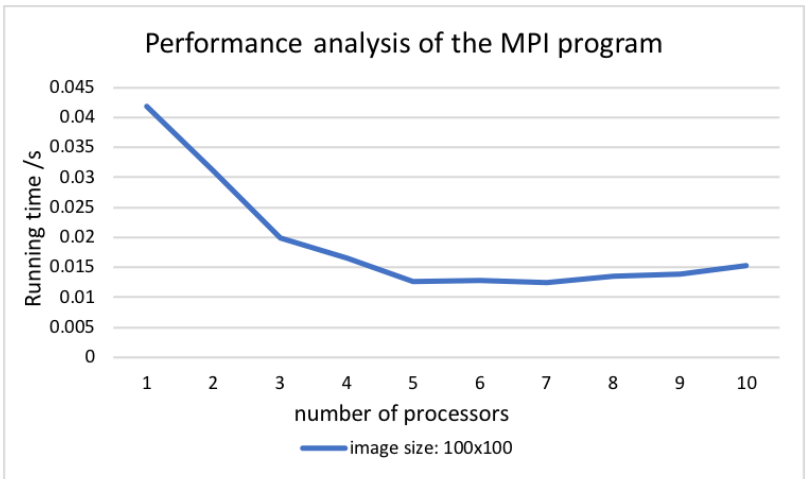


Figure 8. Running time of the MPI program when the image size is 500x500

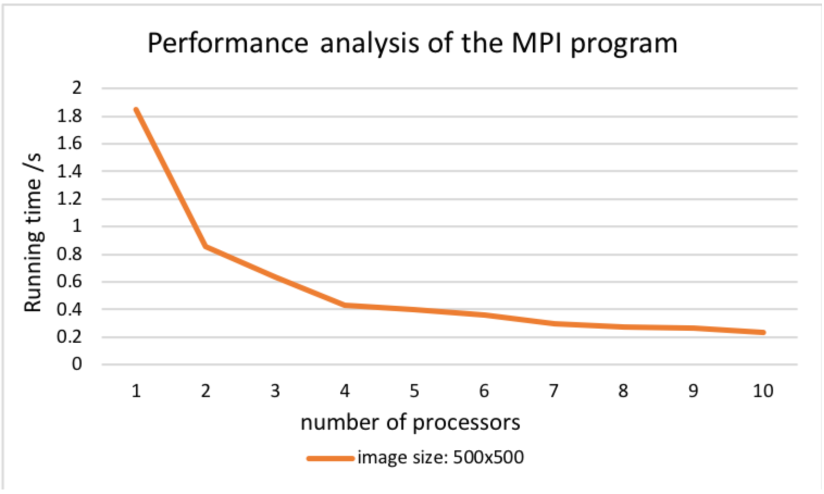


Figure 9. Running time of the MPI program when the image size is 1000x1000

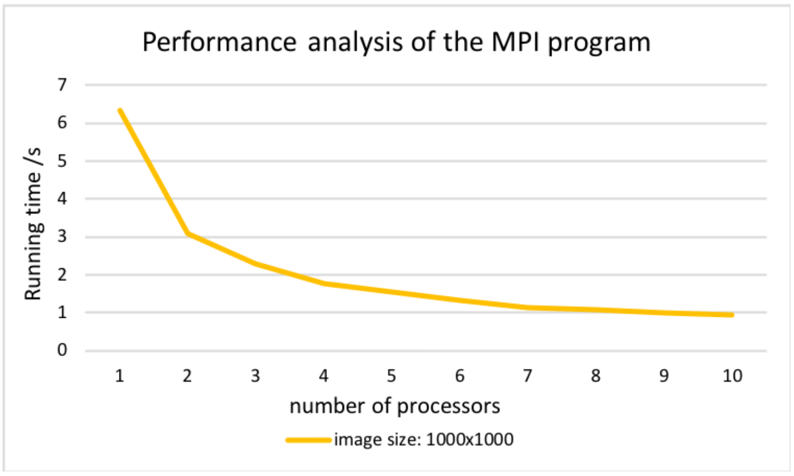


Figure 10. Running time of the Pthread program when the image size is 100x100

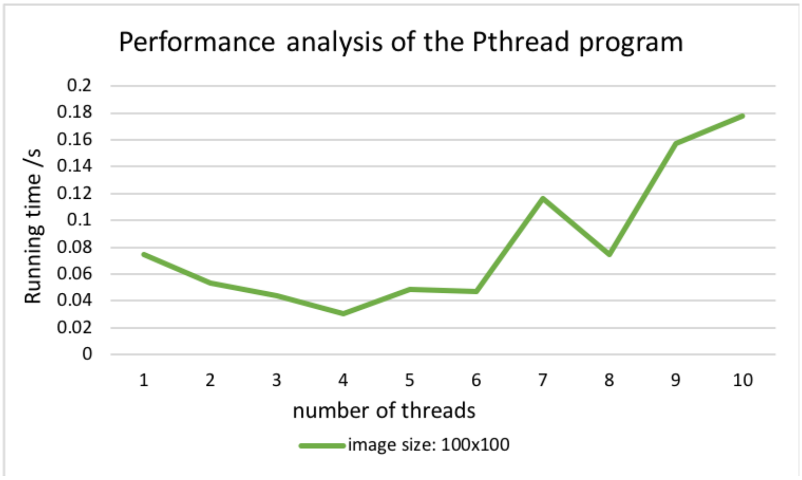


Figure 11. Running time of the Pthread program when the image size is 500x500

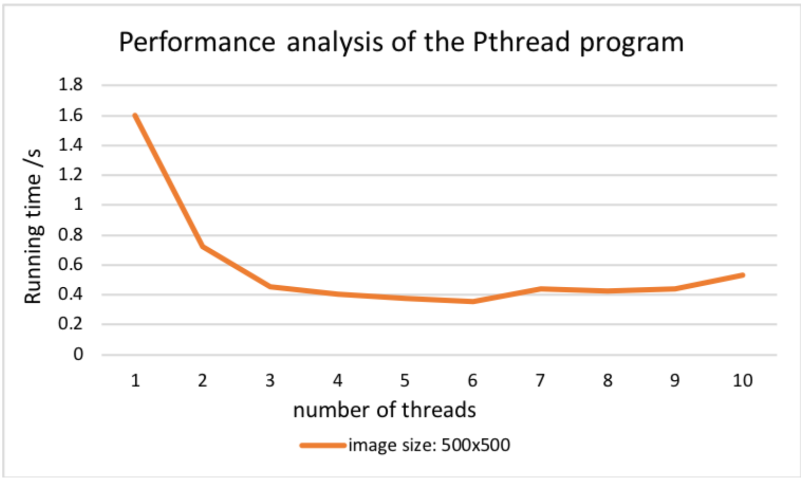
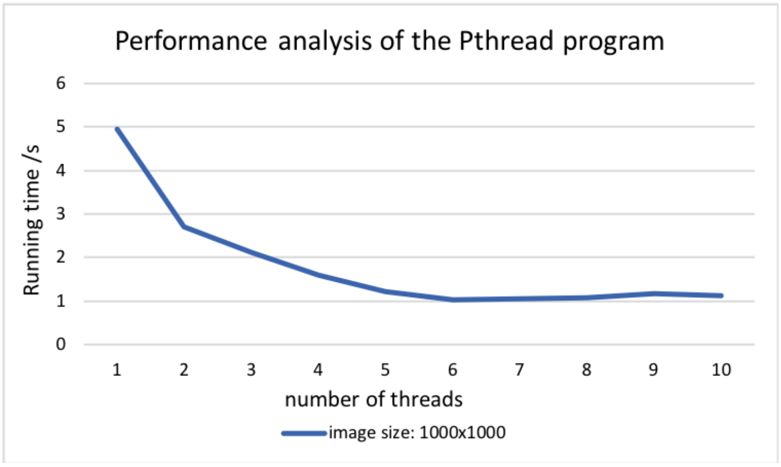


Figure 12. Running time of the Pthread program when the image size is 1000x1000



Notice that in Figure 7, 8 and Figure 9, the number of processors actually maps to the number of processors used in the computation. However, there is one more processor responsible for distributing tasks. Therefore, the total number of processor should be 1 + number of processor used in the computation. For example, if the number of processor in the figure is 1, the total number of processor should be 2. Because the drawing is excluded, the master processor actually has extremely low loading, therefore, the number of processors only used in computation is used in order to conduct fair comparison with other algorithms.

In Figure 8 and Figure 9, it is observed that when the image size is 500x500 and 1000x1000, the running time of the programs declines monotonically along with increasing number of processors. However in Figure 7, the running time declines monotonically only when the number of processors is less than 5; if the number of processors is larger than 5, the running time increases slightly as the number of processors increases. In my assumption, the slight increment of the running time results from the increased communication time between processors. Because the problem size is small, the time saved on computation is trivial as the number of processors increases, the overhead of the communication time is heavy. Therefore, the communication time's increment surpasses the computation time's decrement. As the problem size increases, the overhead of the communication become trivial; the amount of time saved on computation is essential, which results in performance gain.

From Table 1 and Figure 8, 9 and Figure 10, the running time of the MPI program slightly surpasses the running time of the sequential program when the number of processors used in the computation is 1. This is because the communication required in the MPI program, which slightly deteriorates the performance. However, when the number of processors is larger than 2, the running time of the MPI program has significant improvement over the sequential program no matter under what image size and the number of processors.

Figure 11, 12 and Figure 13 demonstrate the running time of the Pthread program. In Figure 11, when the image size is 100x100, the running time decrease with increasing number of threads only when the number of threads is less than 4. The running time increases significantly when the number of threads is larger than 4. And the Pthread program has worse performance than the sequential program no matter how many threads are used when the image size is 100x100.

In Figure 12, when the image size is 500x500, the running time of the Pthread program reaches the global minimum when the number of threads is 6. However, when the number of threads is larger than 6, the running time slight increases with increasing threads. However, when the number of threads is larger than 2, the Pthread program would have better performance than the sequential program; when the number of thread is larger than 3, the Pthread program would have significant improvement over the sequential program.

In Figure 13, when the image size is 1000x1000, the running time of the Pthread program still reaches the global minimum when the number of threads is 6, and still has a slightly increasing trend when the number of threads is larger than 6. And the acceleration effect becomes significant when the number of threads is larger than 2.

Compared Pthread implementation and MPI implementation. When the image size is 100x100, the Pthread program always has worse performance than the MPI program. However, it is interesting that the Pthread program has better performance than the MPI program when the number of threads/processors is less than 3 and the image size is either 500x500 or 1000x1000. It is because directly reading from the memory is faster than distributing the arrays using MPI_Send() and MPI_Recv(). However, the MPI program surpasses the Pthread program when the number of threads/processors is larger than 3. It can be explained that the lock and unlock operations in the Pthread programs would possibly stall all other threads, and as the number of threads increases, the lock and unlock operations would become frequent, and the possibility that the lock and unlock operations stall other threads become higher. For the MPI program, although communication between two processors may stall the other processors, the communication is much less frequent than the lock and unlock operations in Pthread programs when many threads/processors are used.

Conclusion

In this experiment, the parallel heat distribution simulation was successfully implemented using MPI and Pthread. The performance analysis shows that the MPI program and the Pthread program has significant acceleration effect when the image size is large and the number of processor/thread is larger than 2. The Pthread program has better performance than the MPI program under a small number of processor/thread. However, the MPI program surpasses the Pthread program under a large number of processor/thread.

Experience

In the Pthread program, it is essential to reduce the unnecessary lock and unlock operations because lock and unlock operation will stall the other threads. [redacted]

In the MPI program, there is a small trick that can boost up the program when the image size is small. In MPI program, there is one processor acts as a master processor and distributes tasks to other processors. It is easy to make the processors except for the master processor to have the same loading [redacted]

[redacted]

Pthread works well when the number of threads is small, for example, less than 4. [redacted]