Assignment 4 Heat Simulation
CSC4005 Distributed and Parallel Computing

Cui Yuncong
School of Data Science
The Chinese University of Hong Kong, Shenzhen
118010045@link.cuhk.edu.cn

1. Introduction

The heat simulation generates a model in which there are four walls and a fireplace. The temperature of the wall is 20 °C, and the temperature of the fireplace is 100 °C. In this assignment, we make use of Jabobi iteration to compute the temperature inside the room and plot temperature contours at 5 °C intervals using Xlib.

Jacobi Iteration is invented to solve $n \times n$ near 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:

$$x^{k+1} = D^{-1}(b - Rx^k) \tag{1}$$

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

$$x_i^{k+1} = \frac{1}{a_{ii}}(b_i - \sum_{j \neq i} a_{ij} x_j^k), i = 1, 2, 3, ..., n \tag{2}$$

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:

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}| \tag{3}$$

2. Method

The below figures illustrate the output of the design of the MPI method, the Pthread method, OpenMP method and MPI-OpenMP Method.

2.1. MPI Method

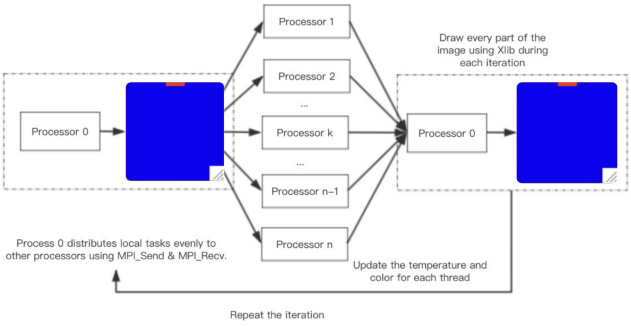

Figure 1: The Program Flow of MPI Heat Simulation

2.2. Pthread Method

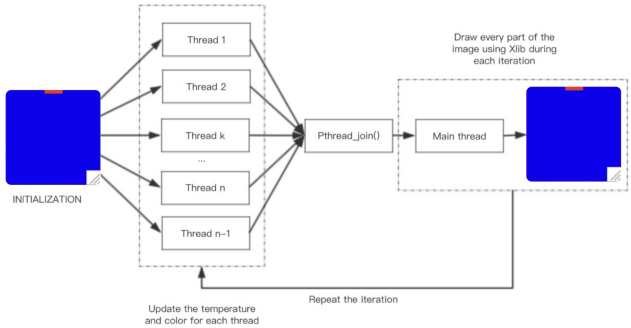

Figure 2: The Program Flow of Pthread Heat Simulation

2.3. OpenMP Method

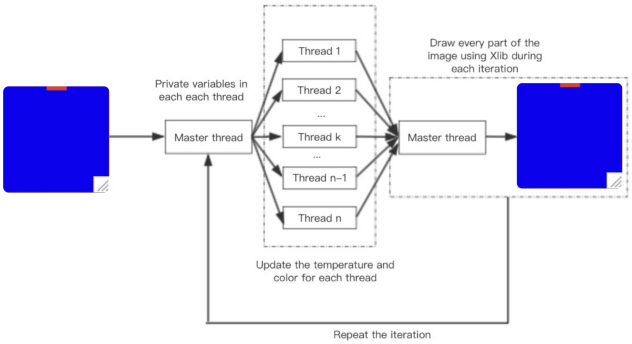

Figure 3: The Program Flow of OpenMP Heat Simulation

2.4. MPI-OpenMP Method

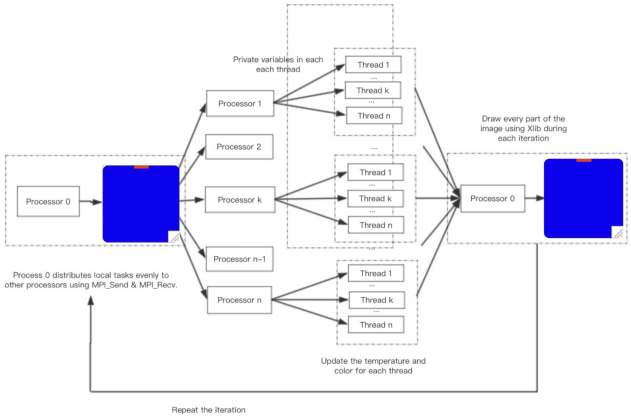

Figure 4: The Program Flow of MPI-OpenMP Heat Simulation

3. Experiment

3.1. Execution Steps

The default setting is not to show the result image. If the image needs to be printed, please change the bool variable PIC from false to true in each source code.

3.1.1 Sequential Method

Type "**`g++ hw4_sequential.cpp -lX11 -lm -o hw4_sequential.out`**" to compile the sequential method. Type "**`./hw4_sequential.out`**" to execute the sequential method.

3.1.2 MPI Method

Type "**`mpic++ hw4_mpi.cpp -lX11 -lm -o hw4_mpi.out`**" to compile the Pthread method. Type "**`mpirun -n 2 ./hw4_mpi.out`**" to execute the MPI method.

3.1.3 Pthread Method

Type "**`g++ hw4_pthread.cpp -lX11 -lpthread -lm -o hw4_pthread.out`**" to compile the Pthread method. Type "**`./hw4_pthread.out`**" to execute the Pthread method.

3.1.4 OpenMP Method

Type "**`g++ hw4_openmp.cpp -lX11 -fopenmp -lm -o hw4_openmp.out`**" to compile the Pthread method. Type "**`./hw4_openmp.out`**" to execute the OpenMP method.

3.1.5 MPI-OpenMP Method

Type "**`mpic++ hw4_mpi_openmp.cpp -lX11 -fopenmp -lm -o hw4_mpi_openmp.out`**" to compile the Pthread method. Type "**`mpirun -n 2 ./hw4_mpi_openmp.out`**" to execute the MPI-OpenMP method.

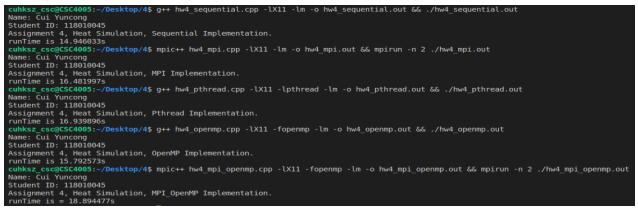

Figure 5: Test Result in Command Lines

3.2. Data Analysis

Noticed that the display of the image is excluded from the final evaluation since the drawing process is extremely timeconsuming on the cluster server and time spent on the computation (or communication) is quite trivial compared with the running time of the drawing. In this case, we will not take the drawing time into consideration. The performance analysis is conducted in two dimensions, varying in body size and the number of processors / threads.

It should be highlighted that in all the figures below and below, the number of processors in MPI method means the number of processors used in computation since there is a master processor which is responsible for sending and receiving the the messages to or from all the slave processors. There exists a processor 0.

The actual number of processors is $(n+1)$. Since the drawing step is not taken into account, the time spent on the master processor is quite trivial. Thus, we only use the number of slave processor to conduct the performance analysis.

In general these results exhibit reasonable speedup as the number of participating processes increases. The estimated execution-time is O(nm) for serial program. A parallel implementation can be estimated by O(nm/p) where n and m are size and p is number of processes.

3.2.1 Small Image Size: 100×100

As demonstrated in Figure 6, when the image size is small (100×100), the performance of pthread method is worse than the MPI method when number of thread/processors 1-16. When the number of processors grows, the contrast is even distinct. The Pthread method, in this case, have a overall worst performance. It is because when the image size is small, pthread program has only 1 processor, the time spent on loading and storing thread information is quite large. OpenMP program has the best performance over the other methods.The reason why OpenMP has a respectively better performance under small size of image is that it is based on the shared-memory system while MPI is based on the massage-passing system. The sharedmemory system does not require that massage passes from one processors to another. They pass the massage by pointer which save a lot of time under small size.

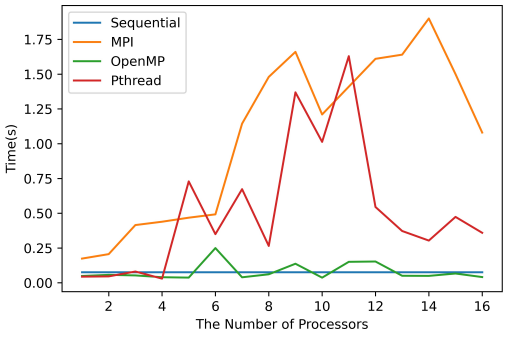

Figure 6: Performance Analysis of the MPI, Pthread, OpenMP and Sequential Method (Small Image Size)

3.2.2 Medium Image Size: 500×500

As demonstrated in Figure 7, when image size becomes medium(500×500), both the Pthread and OpenMP have a better performance than Sequential method no matter the number of the processors. When the number of processors smaller than 8, MPI also has a better performance than sequential method. In this case, Pthread have the best performance over MPI and OpenMP method. OpenMp have a better performance than MPI. When the number of processors/threads increases, both of Pthread and OpenMP have a respective increase in performance. However, it seems that when the number of processor grows, the MPI programs will have a worse performance. This is because my design spent too much time on communication which leading to a time increment when the number of processor grows.

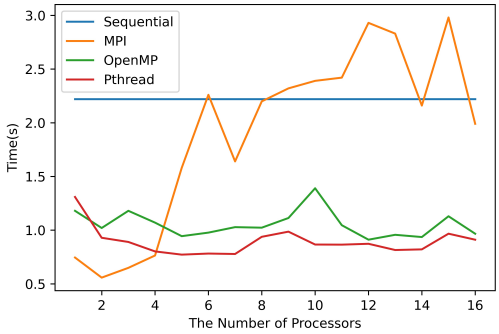

Figure 7: Performance Analysis of the MPI, Pthread, OpenMP and Sequential Method (Medium Image Siz)

3.2.3 Large Image Size: 1000×1000

As demonstrated in Figure 8, when image size becomes large(1000×1000), both the Pthread and OpenMP have a better performance than Sequential method no matter the number of the processors. It is because directly reading from the memory is faster than MPI_Send and MPI_Recv(). When the number of processors smaller than 13, MPI also has a better performance than sequential method. Comparing with Figure 7 where shows that under medium image size, only when the number of processor is smaller than8, MPI will have a better performance. There is still a performance improvement for MPI method, however, since the design is not so perfect, the running time of MPI is still increasing when the number of processor increases.

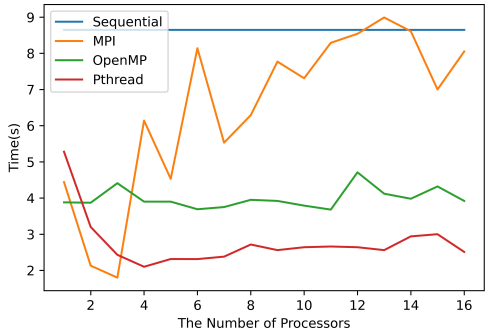

Figure 8: Performance Analysis of the MPI, Pthread, OpenMP and Sequential Method (Large Image Size)

3.2.4 Image Size Comparision

In Figure 9, the running time of MPI, OpenMP, Pthread, Sequential and MPI-OpenMP programs all grow when the image size grows. Noticed the value of x-coordinate is not proportionally increase. The running time of sequential method show like a polynomial curve. The running time of sequential method is in proportion to the image size.

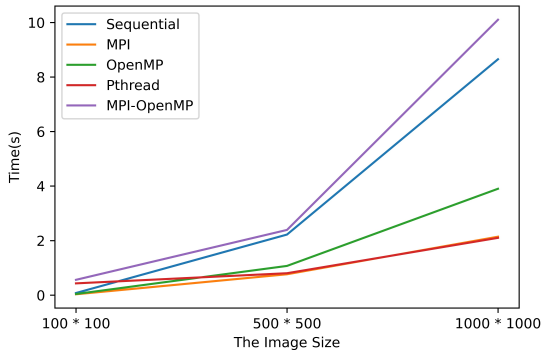

Figure 9: Performance Analysis of the Sequential, MPI, OpenMP, Pthread and MPI-OpenMP (4 Processors)

3.2.5 MPI and MPI-OpenMP

Figure 10 demonstrates the performance of the MPI and MPI-OpenMP programs when the image size is 100×100, 500×500 and 1000×1000. I fixed the numebr of processors to 4. We can see from the both graphs that when the image size is 100×100 and 500×500, as the number of thread grows from 1 to 6, the performance of MPI-OpenMP programs are all worse than only MPI program, but we can see that wen image size us large, as the thread number keeps growing, the running time of the MPI-OpenMP programs start to decrease but there is still not a huge gap between. As just mentioned before, the reason is that my design of MPI program sepend a lot of time of communication.

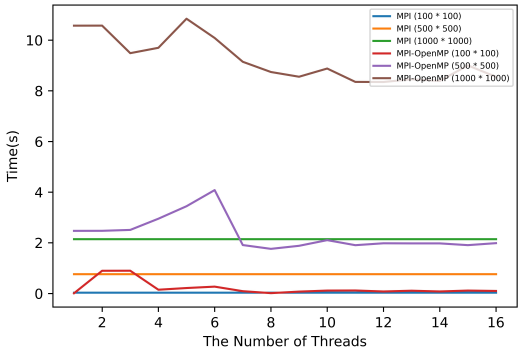

Figure 10: Performance Analysis of the MPI and MPI-OpenMP Method (Image Size: 100×100, 500×500 and 1000×1000 with 4 Processors)

3.2.6 OpenMP and MPI-OpenMP

Figure 11 demonstrates the performance of the OpenMP and MPI-OpenMP programs when the image size is 100×100, 500×500 and 1000×1000. Still, I fixed the numebr of processors to 4. The MPI-OpenMP programs conduct a worse performance than the OpenMP prorgam. But compared with the MPI only program (Figure 10), we can see that the gap is smaller. This is because OpenMP has a better performance than MPI in this case.

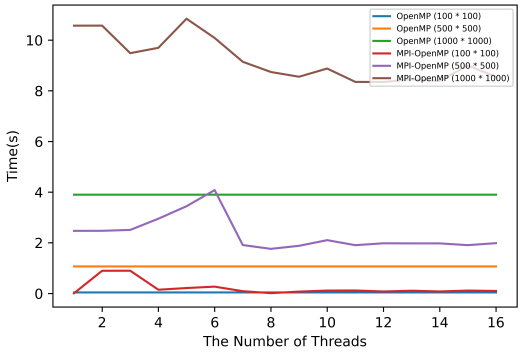

Figure 11: Performance Analysis of the OpenMP and MPI-OpenMP Method (Image Size: 100×100, 500×500 and 1000×1000 with 4 Processors)

4. Code Implementation

The Figure 12 demonstrates the implementation of initialization of 20 colors.

```
for (int i=0; i<20; ++i)
{
    double w = double(i)/20;
    color[i].green = 66*64+150*64*(1-w);
    color[i].red = 65535*w;
    color[i].blue = 65535*(1-w);
    color[i].flags = DoRed | DoGreen | DoBlue;
    XAllocColor(display, default_cmap, &color[i]);
}
```

Figure 12: Initialization of 20 Colors

The Figure 13 demonstrates the implementation of iteration of each time.

```
void temperatue_iterate(double *map)
{
    int i, j, d;
    for (i = 0; i < RES_X; i++)
    {
        for (j = 0; j < RES_X; j++)
        {
            int cnt = 0;
            tempmap[i * RES_X + j] = 0;
            for (d = 0; d < 4; d++)
            {
                if (legal(i+dx[d], RES_X) && legal(j+dy[d], RES_X))
                {
                    tempmap[i*RES_X + j] += map[(i+dx[d])*RES_X + (j+dy[d])];
                    cnt ++;
                }
            }
            tempmap[i*RES_X + j] /= cnt;
        }
    }

    for (j = RES_Y/2-RES_Y/12; j < RES_Y/2+RES_Y/12; j++)
        tempmap[j] = 100.0;
}
```

Figure 13: Iteration

5. Conclusion and Improvement

5.1. Conclusion

The Heat Simulation image is displayed by using Sequential, MPI, OpenMP, Pthread and MPI-OpenMP methods. The above performance analysis clarifies why the Pthread method will show bad performance when the image size is small and OpenMP have better performance than MPI. When the body size grows), both of the OpenMP method and Pthread method show a general downward trend of running time. The MPI method has a improvement in performance.

5.2. Improvement

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. 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.