Assignment 4 Heat Simulation
CSC4005 Distributed and Parallel Computing

Chen Yuan 117010038
School of Science and Engineering
The Chinese University of Hong Kong, Shenzhen
117010038@link.cuhk.edu.cn

Abstract

This assignment implements a Heat-Distribution Simulation, using 5 versions – one sequential method and 4 parallel methods, namely the MPI method, pthread method, openmp method, and MPI+OpenMP method. The report will present the principle, codes, as well as comparasion and analysis of performance of different implementations.

1. Introduction

Heat-Distribution Simulation refers to the problem of simulating the temperature distribution in a room. The temperature of a inside point can be calculated as: $h_{i,j} = \frac{h_{i-1,j}+h_{i_1,j}+h_{i,j-1}+h_{i,j+1}}{4}$. Then for each iteration, update all interior points until reaching the maximum iteration or all points are converged.

Here are some information about the heat initial state:

1. A room has four walls and a fireplace. The temperature of the wall is 20°C, and the temperature of the fireplace is 100°C.
2. The fireplace is at the upper middle of the wall square. If the wall has shape $n \times n$, then the fireplace has shape $\frac{n}{3} \times \frac{n}{5}$
3. Initinally, other places except the wall and the fireplace has temperature 0°C. We will use jacobi iteration to find the temperature of thhe whole room.
4. In the visualization, X11 is used to plot temperature contours at 5°C intervals.

The following Figure 1 is a heat color map, where blue in the left side denotes 0°C, and red in te right side denotes 100°C. Figure 2 shows the initial state for the simulation with display size 200 × 200. Figure 3 is shows the final state for the simulation with maximum iteration equals to 800 and threshold for converging equals to 0.01.

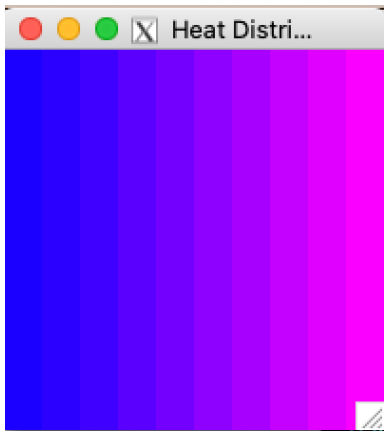

Figure 1. color map from 0°C to 100°C by X11

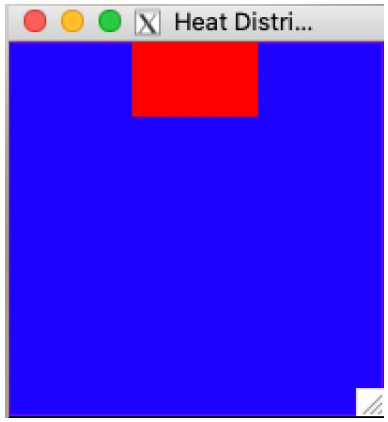

Figure 2. Initial state for size 200 × 200

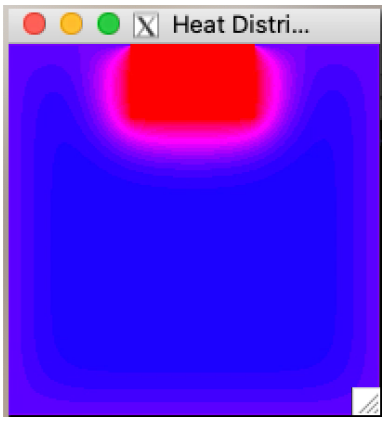

Figure 3. Final state (after 800 iterations) for size 200×200

The implentation details and performance analysis are shown below.

2. Method

For all methods, we use the Jacobi iteration. For parallel methods, we use the number of processes or threads ranging from 1 to 16.

2.1. Sequential

At one time stamp, need to calculate the force of each interior point. Then update the global graph.

Inspired by the lecture, we have the psudocode shown in the following:

map = double[n][n];
map_new = double[n][n];
initialize map and map_new;(walls and fireplaces)
for $iter = 0 \cdots$, $iteration$ do
  cont = false;
  for $i = 1 : n-2$ do
    for $j = 1 : n-2$ do
      $map\_new[i][j] = \frac{map_{i-1,j}+map_{i_1,j}+map_{i,j-1}+map_{i,j+1}}{4}$
      if(!converged(i,j)) cont = true;
    end for
  end for
  for loop again to update points
  if (cont == false) breaak
end for

Data Structures: Data structure to store the points is a 2-D array int[][]. In C++, it is defined by:
a = (double**) malloc(sizeof(double*)*n);
a[i] = (double*) malloc(sizeof(double)*n) for all i = 1...n;

2.2. OpenMP

In the sequential method, there are two for loops in each iteration. One is for calculating the new temperature for each interior point and check convergence, and another is for updating the temperature for each interior point. Then we can add "#pragma omp parallel for" before each for loop.

The flow chart is shown in Figure 4.

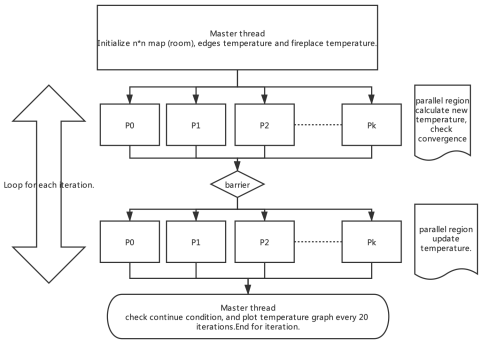

Figure 4. Flow chart for OpenMP implementation

2.3. MPI

In the MPI implemantation, assume there is $n \times n$ room size and k processes including the MASTER node. We use the strip partition in to distribute the parallel jobs.

1. in each process, a complete $n \times n$ size map and map_new is created. This is for the convenience of sending and receving columns, as well as gather the columns to MASTER and draw color map.

2. Attribute the $\frac{n}{k}$ column to each node.The last process will also handle the reminder columns.

3. For each iteration, first calculate the new temperature in each process. update the interior points.

4. Inside each iteration, second do the MPI_SEND (of the new data) to the next column and receive from its next column (if this is not the last process). Do the MPI_SEND to the former column and receive from its former column (if this is not the first process).

5. set a barrier after each loop. Use a global bool variable "global_continue" to check if the points converge. Use MPI_Allreduce to reduce the "cont" variable in each process to "global_continue"

6. every 20 iteration, use MPI_gather to gather all the sub_maps in each process, and draw the color temperature distribution.

Since the MPI version needs extra communication costs, the strip partition is easier in implementation.

The flow chart is shown in Figure 5 .

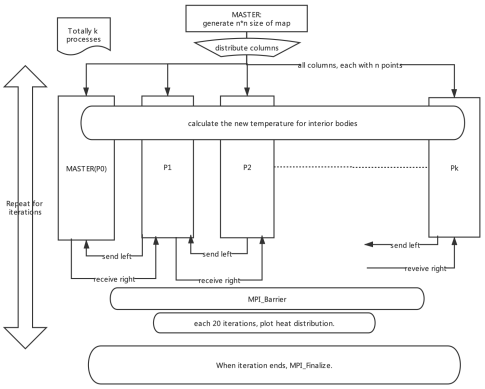

Figure 5. Flow chart for MPI implementation

2.4. Pthread

Since all threads can work in parallel, here we adopt the number of threads ranging from 1 to 16 in the experiment. The program is flexible for different inputs of thread numbers, and room size. Just like the MPI version, we distribute data to each threads with $\frac{n}{k}$ bodies to compute, but the last thread contains $\frac{n}{k} + n\%k$ bodies. In the implementation, pthread_barrier_init(barrier) is used to maintain each thread's pace, in order to finish one iteration. Global variable map, map_new, are used to record and update the temperature for interior points after each iteration. In the middle, two barriers are used. One is after the calulation of map_new and before the updating for k threads, and the other is after each iteration for (k threads+master), which is used to check the data convergence and plot each distribution.

Figure 6 shows the flow chart of Pthread method.

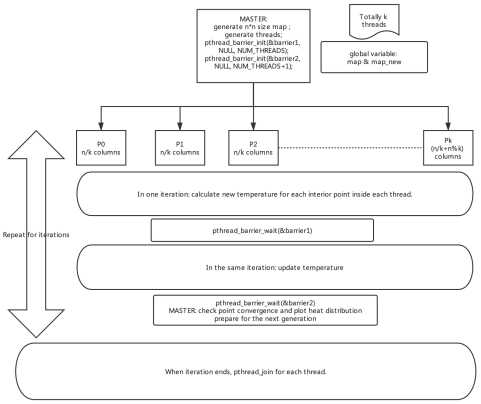

Figure 6. Flow chart for Pthread method

2.5. Command lines

All the heigh and width in the display are 200 by default. For the MPI program, the first instance of main function is room size n. For Openmp programs, the first instance is number of threads(default is 1) and the second instance is room size n. For pthread program, the first instance is number of threads(default is 1), and the second instance is room size n.

For mpi+openmp, instancr is the same as mpi version. Number of threads for each process needs to be specified in the command line.

In order to evaluate the performance namely runtime, the drawing procedure is not included in the parallel programs. However, the "mpi_draw.cpp" indicates a mpi version with the X11 drawing procedure. And a video recording the simulation is attached in the package. For other method, just uncommemt the lines regarding X11 plot.

The sample compile codes are listed below:

#Sequential
g++ ./seq.cpp -o ./seq -lX11
./seq 800
#MPI
mpic++ ./mpi.cpp -lX11 -o ./mpi
mpirun -n 8 ./mpi 800
mpirun -f /home/mpi_config -n 8 ./mpi 800 (for including config)
#Pthread
g++ ./pth.cpp -o ./pth -lpthread -lX11
./pth 8 800
#OpenMP
g++ -o ./openmp ./openmp.cpp -fopenmp
./openmp 8 800
#OpenMP+MPI
mpic++ ./mpi_omp.cpp -lX11 -o ./mpi_omp -fopenmp
OPM_NUM_THREADS=4 mpirun -n 8 ./mpi_omp 800
OPM_NUM_THREADS=4 mpirun -f /home/mpi_config -n 8 ./mpi_omp 800

3. Experiment

Max number of iterations: 500
Threshold for convergence: 0.01

The most important metric here is runtime under different circumtances. Since the X11 drawing procedure is slow, here we only consider the computation runtime, including the data transmiting, and without displaying the graph. The total results are attatched in Appendix A.

One important figure that consolidates all usful information in performance is shown in Figure 7. The red horizontal line in the figure refers to the sequential implementation time (calculated by the average of 5 times).

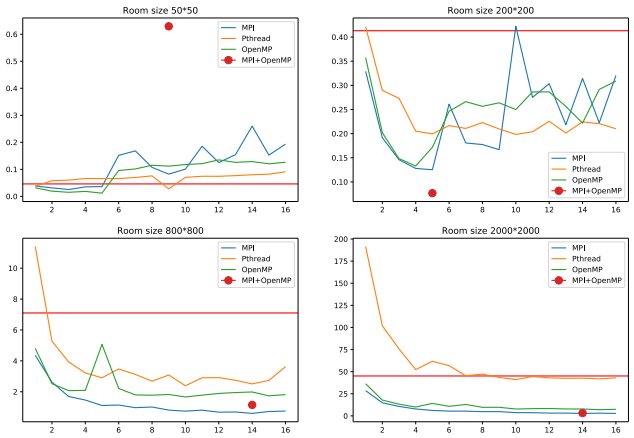

Figure 7. Performance comparasion figure.

Sections below are several analysis based on different evaluation metrics:

3.1. The number of processes or threads used in the program

As shown in Figure 7, there are some outliers in the program. However, number of threads or processes grows, the total time for a program decreases for large data. Meaning that as the nodes for parallel process grows, the total excucation time generally shrinks.

On the figure shown above we can observe that especially for Pthread implementation, as the number of the parallel nodes increases, the excucation time first experiences a sudden decreases for the first 1-4 nodes, then may shrink to a rather low area for more nodes. For parallel nodes exceeding 8, the speed up performance is may not be severe and clear.

However, there is an exception in parallel implementation on small datasets (namely 50×50 size).As the number of processes increases, the total time for overall computation increases, especially for MPI version. The reason behind this may be that since the MPI needs send and receive implementation, the overhead for data transmitting between nodes are huge.

3.2. The output size ranging from 50, 200, 800, 2000

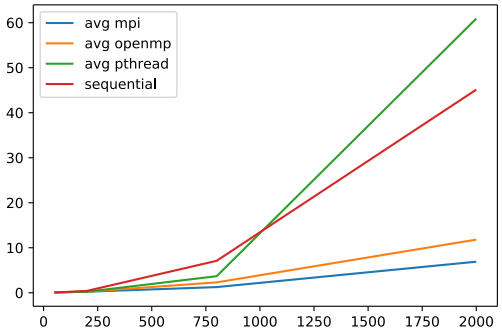

Figure 8. Average running time VS size of room.

The room size greatly influences the time for data processing. As shown in Figure 8, as the number of bodies increases, the running time also increases. It shows a convex function trend and the increase rate in time versus room size is the biggest for sequential work, then for Pthread, then OpenMP, at last MPI. MPI runtime increases really small comparing to the other three methods. Tihs is also because that MPI runtime is small.

3.3. MPI vs Sequential vs Pthread vs OpenMP vs MPI+OpenMP

Since there is no speedup outcome reguarding room size 50×50, we skip this figuresize and do the compare analysisi mainly on figure size larger than 200.

With the study oof different metrics, we can say that the MPI method performs really well, with nearly 4 nodes to be the best option. All three methods would not have significant greater speed up when processors become more than 6-8.

Generally, MPI implementation is good for larger size of data. As data size goes up, it would have a better effciency, and it performs more stable for large number of processors.

OpenMP could be the second option. It performs not bad in all cases, and would be stable for different number of nodes. Actually for OpenMP method, we can use the simplified version of force computing as mentioned above, to avoid exhausting calculation. However, in real test, it results in a larger running time than normal method. Pthread method encounter a big drop from 1 to 4 threads in general, which is more obvious than other methods. Sometimes in lower number of nodes, its running time would be larger than that of sequential running time. However, as the input size increases, it may increase more in running time than other methods.

For MPI+OpenMP method, we choose the number of threads with best MPI performance in each data size to perfprm MPI+OpenMP. An iteration through sub_threads ranging from 2 to 11 is conducted. Then we find the one with least execution time. As seen in the figure, the speedup for MPI+OpenMP is clear for room size 200. When room size increases, the speedup effect is unclear. This may because that the performance in this ssh tunnel is not stable.

Overall, the parallel method performs better than sequential method.

Actually in the ssh tunnel provided by the school, the overhead is large, and for different running times, the optput may be different as well, since all students in this course are using it. Hence, I use another ssh tunel with better configuration to run the same experiments. Configuration is the same as well.

The results are shown in Fig 9. The output is labeled as 'seq_other', 'mpi_other'.....

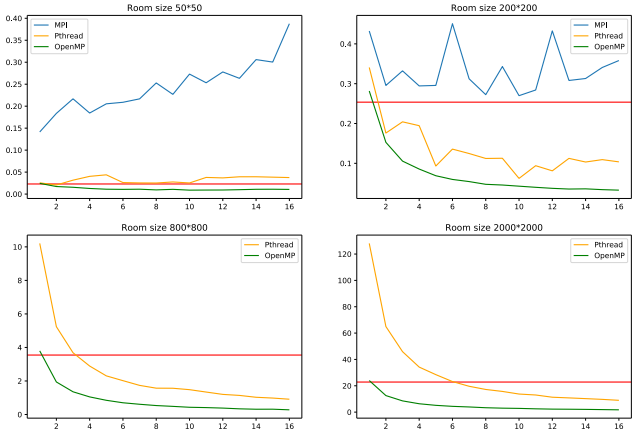

Figure 9. Performance comparasion figure in another ssh tunel.

We can see that the 'other' ssh tunel has better performance than the course ssh tunel. For example, the time for sequential code is nearly the half of that in the school tunel. The MPI implementation here is not satisfying, and even has longer running time than sequential. This is because the new ssh tunel doesn't not have mpi configuration. So we will not compare the performance for mpi.

For openMP and pthread, we can get a better speedup. There are less fluctuations, and nearly no fluctuations when room size exceeds 800. OpenMP gets a better speedup performance than pthread. And as the number of threads increase, the execution time gets smaller for all cases.

4. Conclusion and Reflection

In this experiment, 4 parallel versions of heat simulation is conducted with pthread, MPI, OpenMP and MPI+OpenMP approach. The results shows that generally as the number of threads or processes increases, the total time for the problem will decrease. After 8 nodes, the performance would not have a big speed up. And the number of bodies determines generally how much time a sequential version could be used. As the size of figure increases, the time needed increases as well. The MPI+OpenMP has speed up effect, and is most clear in room size 200.

Through this project, I understand more about the implementation details of different parallel methods, and trained my X11 implementation as well. Since one task is only limited to 5 min on the server, it would be better if I can find the running time for larger room size if time permits. And since there are tpp many people using the platform, theoutcome is not stable.

A. Experiment Results

The total results are listed in Figure 10 and Figure 11. Focus on runtime for each case.

name	process/thread	50	200	800	2000
seq	0	0.045936	0.413127	7.101011	45.134802
mpi	1	0.038378	0.328991	4.359669	28.559381
mpi	2	0.031368	0.191594	2.619906	14.802213
mpi	3	0.025205	0.145964	1.69463	10.687443
mpi	4	0.03508	0.127901	1.462771	7.809803
mpi	5	0.036082	0.125214	1.110327	6.136833
mpi	6	0.15165	0.261024	1.139642	5.373498
mpi	7	0.168207	0.180653	0.970401	5.345913
mpi	8	0.107056	0.177359	1.012685	4.777205
mpi	9	0.082213	0.166798	0.811603	4.794101
mpi	10	0.100907	0.422261	0.74479	3.665131
mpi	11	0.185538	0.275042	0.809297	3.605287
mpi	12	0.125113	0.303424	0.677425	3.155912
mpi	13	0.154429	0.218448	0.691666	3.135354
mpi	14	0.259883	0.314265	0.597605	2.78523
mpi	15	0.152621	0.222248	0.720637	3.054066
mpi	16	0.193261	0.320401	0.750036	2.773009
pth	1	0.037088	0.421163	11.399097	191.476504
pth	2	0.057906	0.289574	5.280006	101.920781
pth	3	0.059964	0.273154	3.931048	75.71914
pth	4	0.066005	0.205096	3.219393	52.349929
pth	5	0.065751	0.199475	2.90457	61.683257
pth	7	0.065791	0.216607	3.47457	56.82923
pth	8	0.069767	0.210781	3.123706	45.42543
pth	9	0.075736	0.222791	2.685245	47.218091
pth	9	0.027575	0.209635	3.082032	43.488691
pth	10	0.070241	0.198356	2.38902	41.041544
pth	11	0.074271	0.203818	2.905217	44.465073
pth	12	0.074137	0.225455	2.914292	43.062081
pth	13	0.076675	0.201308	2.742655	42.48811
pth	14	0.079918	0.224313	2.509794	42.646029
pth	15	0.082016	0.220587	2.734556	41.869154
pth	16	0.090935	0.210094	3.627174	43.011275
openmp	1	0.031552	0.357263	4.80832	36.333179
openmp	2	0.01915	0.201899	2.52056	17.8763
openmp	3	0.014881	0.148325	2.075093	13.231567
openmp	4	0.018295	0.132881	2.088999	10.053452
openmp	5	0.011678	0.171667	5.066599	14.08706
openmp	6	0.095587	0.24632	2.214168	10.770317
openmp	7	0.101368	0.266229	1.795281	12.941754
openmp	8	0.11511	0.256504	1.777571	9.654523
openmp	9	0.111809	0.263809	1.825082	9.727031
openmp	10	0.117585	0.249862	1.661813	7.744833
openmp	11	0.120994	0.286369	1.78172	8.187104
openmp	12	0.134959	0.286369	1.891435	8.322094
openmp	13	0.125718	0.256342	1.949271	7.821192
openmp	14	0.128667	0.221857	1.986046	7.784668
openmp	15	0.120299	0.291448	1.735821	6.984182
openmp	16	0.126306	0.308996	1.813903	7.443623
mpi_openmp	2	0.69232	0.089289	1.472229	3.556283
mpi_openmp	3	0.651056	0.076906	1.198945	3.445166
mpi_openmp	4	0.79767	0.080002	1.26585	3.787171
mpi_openmp	5	0.686964	0.098205	1.279274	3.573506
mpi_openmp	6	0.630566	0.097485	1.179419	3.201386
mpi_openmp	7	0.629427	0.098781	1.201625	3.398438

Figure 10. Overall result1

mpi_openmp	8	0.672448	0.076656	1.150121	3.467469
mpi_openmp	9	0.700283	0.100254	1.330529	3.475696
mpi_openmp	10	0.756084	0.080628	1.267463	3.468727
mpi_openmp	11	0.629618	0.091922s	1.289128	3.327067
mpi_openmp_cinfig	0	9	5	14	14
seq_other		0.022993	0.253616	3.54645	22.830088
pth_other	1	0.025557	0.340779	10.206887	128.08453
pth_other	2	0.020733	0.176042	5.237697	64.965859
pth_other	3	0.031664	0.204214	3.684496	45.903809
pth_other	4	0.040343	0.19454	2.891002	34.191545
pth_other	5	0.043814	0.093216	2.304917	28.457156
pth_other	6	0.02619	0.135734	2.018648	23.204602
pth_other	7	0.025369	0.124624	1.73747	19.625342
pth_other	8	0.025208	0.112209	1.570743	17.203664
pth_other	9	0.027575	0.112722	1.566018	15.714109
pth_other	10	0.025318	0.062128	1.481451	13.724631
pth_other	11	0.037884	0.094031	1.339338	13.003978
pth_other	12	0.036826	0.081126	1.199868	11.314184
pth_other	13	0.039373	0.112433	1.138891	10.797513
pth_other	14	0.039407	0.103287	1.027999	10.244553
pth_other	15	0.0385	0.109296	0.980892	9.679338
pth_other	16	0.037585	0.103656	0.911604	8.963937
openmp_other	1	0.024397	0.281641	3.795589	23.980835
openmp_other	2	0.017181	0.152669	1.940715	12.555302
openmp_other	3	0.015488	0.105565	1.35473	8.529043
openmp_other	4	0.012784	0.085618	1.050135	6.353079
openmp_other	5	0.011083	0.068843	0.847442	5.149472
openmp_other	6	0.010739	0.059528	0.699875	4.385579
openmp_other	7	0.011085	0.054172	0.610761	3.919841
openmp_other	8	0.009532	0.047291	0.533923	3.33335
openmp_other	9	0.010782	0.045512	0.485009	3.005876
openmp_other	10	0.008955	0.042611	0.431327	2.832249
openmp_other	11	0.009173	0.039772	0.412112	2.505817
openmp_other	12	0.009281	0.037188	0.382403	2.274381
openmp_other	13	0.010022	0.035412	0.338594	2.196625
openmp_other	14	0.010898	0.035904	0.314518	2.076738
openmp_other	15	0.01102	0.03391	0.31669	1.915717
openmp_other	16	0.010635	0.032637	0.279096	1.743682
mpi_other	1	0.141335	0.432304	3.984218	
mpi_other	2	0.183605	0.295639		
mpi_other	3	0.216739	0.332272		
mpi_other	4	0.184405	0.294404		
mpi_other	5	0.205369	0.295652		
mpi_other	6	0.208869	0.450611		
mpi_other	7	0.21666	0.312253		
mpi_other	8	0.25289	0.272444		
mpi_other	9	0.226845	0.343017		
mpi_other	10	0.273005	0.269921		
mpi_other	11	0.25327	0.284163		
mpi_other	12	0.277919	0.43267		
mpi_other	13	0.263352	0.308107		
mpi_other	14	0.305991	0.313017		
mpi_other	15	0.300359	0.340687		
mpi_other	16	0.387609	0.358144		

Figure 11. Overall result2

B. Codes

All codes, video, analysis codes, can be found in the folder attached. On the server, the codes are just in /home/117010038.