Assignmen4 Report

CSC4005
Prof. Yeh-Ching Chung
TA: Hongliang Zhu & Peipei Zhu

Yu Chen
115010124
6 December 2018

Problem Descrption

There is a room of 50 ft in height and width, at the temperature of 20°C. A fire place is in the middle of the top wall of the room. It has length of 20 ft and temperature of 100°C. The wall is at 20°C constantly and will not be heated by the fireplace. Simulate the heat distribution when the temperature is balanced. Draw it in 5°C temperature contours.

Instruction

This assignment implemented three version of Heat distribution simulation program with graphical output. One is sequential program. The other two are parallel program using multiprocess by MPI library and multithread by pthread library. The complied executable file is in the cluster at direction assignment4 under my user root, named as `Heatseq`, `Heatpth`, and `Heatmpi`. For MPI program, use command `mpiexec -np 4 ./Heatmpi` to run the program with 4 process. The number of process can be changed by the parameter `-n`. The Then re-build by `mpicc`. The command should include x, y, iteration number, and error.

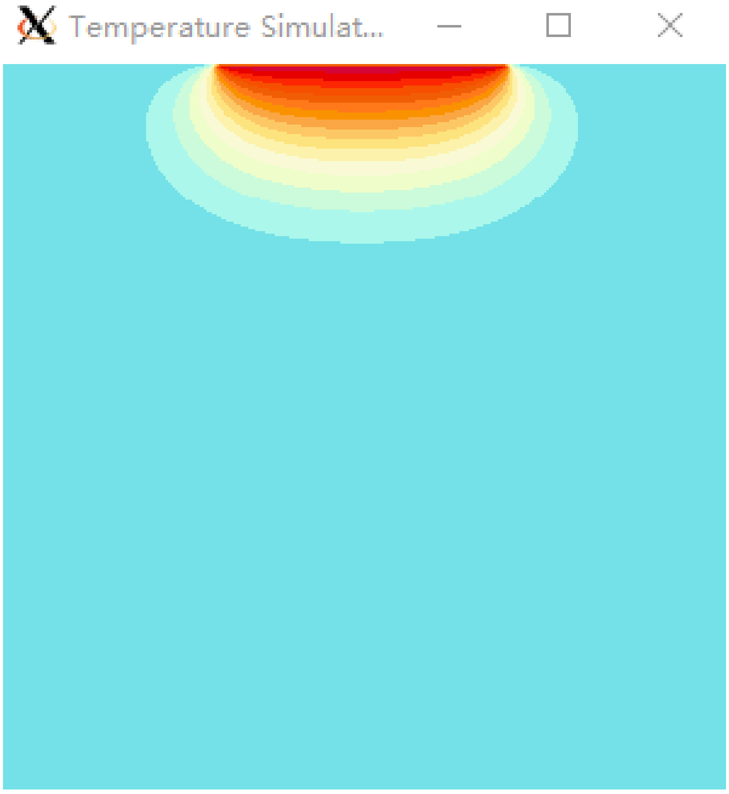

Figure 1: A sample simulation screen shot.

Fig. 1 is a sample output with the resolution of $300 * 300$ simulated with a image output each 100 iteration, the program runs 1000 iterations if not converge, the convergence tolerance error is 1°C. The simulation is not included the time, only the step concept used. Unfortunately,there is some problem in for mpi version to implement the termination condition, so the mpi version should only imput x, y and iteration number, the program terminate only when iteration finished. The running time and terminated iteration of the program will be shown in terminal when the calculation of the picture finished.

Design

Finite difference method

Points number from 1 for convenience and include those representing the edges. Each point will use the equation:

$$x_i = \frac{x_{i-1} + x_{i+1} + x_{i-k} + x_{i+k}}{4} \tag{1}$$

It could be transform as a linear equation containing the unknowns x_{i-k}, x_{i-1}, x_{i+1}, and x_{i+k}. Known as finite difference method, also can solve Laplace's equation. It is fine for this project. The benefit of the algorithm is obvious - easy to implement.

Performance improving algorithm

The termination condition algorithm can reduce calculation time within a tolerance error:

$$T_{error} = \sum_{i,j} |T_{k,i,j} - T_{k-1,i,j}| \tag{2}$$

There is also some other method to reduce the calculation time, like increment algorithm. However, increment algorithm can only save the compute time in first several iteration, if the increment is too slow, there may occur errors totally unacceptable.

Color

The default color provided by sample code is random generated. It cannot give a direct sense of temperature as well as the gradient. I tried some of the color set like Fig. 2 shows. It is not that good looking. Finally, I referred to the standard color for temperature established by China Whether Department.[4]

MPI design

The job assigned to slaves is divided by block. Each slave receive the whole temperature array then calculate their part.

The calculation job for each body is almost the same. So, the master assigned the job equally to each slaves. The slave processes update temperature of their area in the job for each iteration.

After all the slaves finished its job, they send back their job in a new body array structure $field->t$ back to master by $MPI\_Allgather$. The master process draw the frames on screen frame by frame.

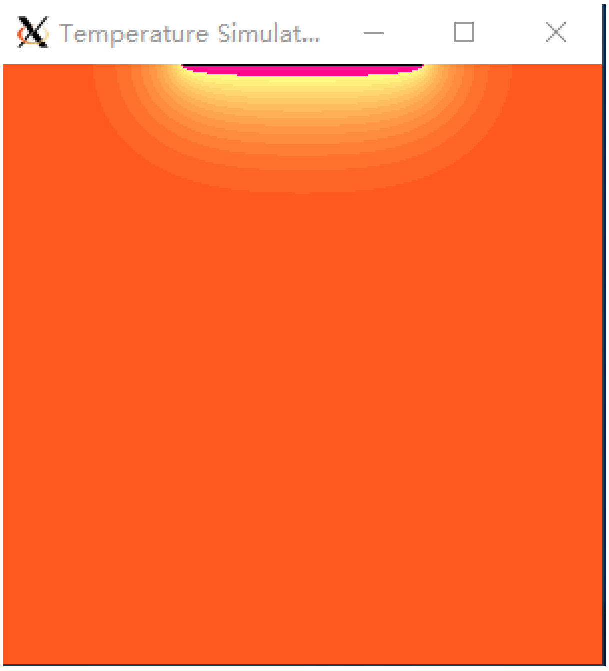

Figure 2: A sample color drawing screen shot.

Pthread design

First, initialize the pthread. In this task `pthread_mutex` is used to insure data correctness. Two arrays are controlled by mutex lock, one is $subThreadWakeUp$ and the other is $subThreadFinished$. These two holds the global information of the block update statue in the frame.

The block size is fix so the thread can locate its area if it knows the start point. In each iteration, the threads calculate its block parallel.

Experiment & Analysis

The test uses 4 process or threads, which is representative number for multitasks. Experiment compares three methods of implementation.

	200	400	600	800	1000	2000
sequential	1.44	4.55	9.86	17.26	27.29	104.25
MPI	1.50	2.42	4.01	6.02	8.96	30.58
pthread	3.84	7.61	/	11.87	17.66	62.12

Table 1: 4process/threads, 1000 steps for different number of body.

The table. 1 is the performance test for three versions program. The process/thread used in MPI and pthread is 4. From table we can see that the MPI takes great advantage when iteration number reaching 2k. The speed up factor is increasing with iteration number, which is 3.41 for 2k iterations. the 600 iteration of pthread lost. The pthread program takes advantage when iteration number reaches 800. Its speed up factor is 1.68 when body number reaches 2k. The performance should further increase for a lager number.

Experience

Algorithm Improvement

The mpi version sends all the temperature infomation to slaves. Theoretically, there existing algorithm decreasing the message size sending to slaves.

X11 Delay

It is always that the figure printing time is much longer than running time in pthread program of this assignment. I cannot find out the reason for this. The time stamp of pthread is right. Compare with the running time of sequential and MPI program, the running time for pthread is also reasonable. Maybe there is some problem when X11 using multi-threads or I miss some sets in X11.

Server load

I did my experiments nearing the deadline time. So the pressure of the server was quite large. What worse, users could only run the program in a single node. Due to the CPU load for huge number of parallel tasks. The test time was not that stable. The analysis part is preferred do in another time for a better result.

Source Code

The assignment is demo in a server. The source code for the submission time is compressed into a zip file and submitted to the Blackboard. A plaintext version is in below.

There are two head file for the assignment, display.h and models.h The first one is for X11 drawing and the second one is for temperature model structures and initialing setup.

```
/* Initial heat distrubution program */


#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <X11/Xos.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <stdlib.h>
#include "models.h"
#include "const.h"

Window          win;            /* initialization for a window */
unsigned
int             width, height,                  /* window size */
border_width,                   /*border width in pixels */
idth, display_height,  /* size of screen */
screen;                         /* which screen */

char            *window_name = "Temperature Simulation", *display_name = NULL;
GC              gc;
unsigned
long            valuemask = 0;
XGCValues       values;
```

```
Display          *display;
XSizeHints       size_hints;
Pixmap           bitmap;
FILE             *fp, *fopen ();
Colormap         default_cmap;
XColor           color[256];

int temperatue_to_color_pixel(double t)
{
return color[(int)(t/5.0f)].pixel;
}

void XWindow_Init(TemperatureField *field)
{
XSetWindowAttributes attr[1];

/* connect to Xserver */

if (  (display = XOpenDisplay (display_name)) == NULL ) {
fprintf (stderr, "drawon:␣cannot␣connect␣to␣X␣server␣%s\n",
XDisplayName (display_name) );
exit (-1);
}

/* get screen size */

screen = DefaultScreen (display);

/* set window size *///XFlush (display);

width = field->y;
height = field->x;

/* create opaque window */

border_width = 4;
win = XCreateSimpleWindow (display, RootWindow (display, screen),
width, height, width, height, border_width,
BlackPixel (display, screen), WhitePixel (display, screen));

size_hints.flags = USPosition|USSize;
size_hints.x = 0;
size_hints.y = 0;
size_hints.width = width;
size_hints.height = height;
size_hints.min_width = 300;
size_hints.min_height = 300;

XSetNormalHints (display, win, &size_hints);
XStoreName(display, win, window_name);

/* create graphics context */

gc = XCreateGC (display, win, valuemask, &values);

```

```
default_cmap = DefaultColormap(display, screen);
XSetBackground (display, gc, WhitePixel (display, screen));
XSetForeground (display, gc, BlackPixel (display, screen));
XSetLineAttributes (display, gc, 1, LineSolid, CapRound, JoinRound);

attr[0].backing_store = Always;
attr[0].backing_planes = 1;
attr[0].backing_pixel = BlackPixel(display, screen);

XChangeWindowAttributes(display, win, CWBackingStore | CWBackingPlanes | CWBackingPi

XMapWindow (display, win);
XSync(display, 0);

/* create color */
int i;
for (i=0; i<20; ++i)
{
color[i].green = 10000 + i * 3200;
color[i].red = 65535;
color[i].blue = 2000*i;
color[i].flags = DoRed | DoGreen | DoBlue;
XAllocColor(display, default_cmap, &color[i]);
}
}

void XResize(TemperatureField *field)
{
XResizeWindow(display, win, field->y, field->x);
}

void XRedraw(TemperatureField *field)
{
int i, j;
for (i=0; i<field->x; ++i)
for (j=0; j<field->y; ++j)
{
XSetForeground(display, gc, temperatue_to_color_pixel(field->t[i][j]));
XDrawPoint (display, win, gc, j, i);
}
XFlush (display);
}
```

```
#ifndef _MODELS
#define _MODELS

#include <memory.h>
#include <stdlib.h>
#include "const.h"

#define legal(x, n) ( (x)>=0 && (x)<(n) )

typedef struct TemperatureField
{
int x, y;
```

```
double **t;
double *storage;
}TemperatureField;

void deleteField(TemperatureField *field);

void newField(TemperatureField *field, int x, int y, int sourceX, int sourceY)
{
TemperatureField temp = *field;
field->storage = malloc( sizeof(double) * x * y );
field->t = malloc( sizeof(double*) * x );
field->x = x;
field->y = y;
int i, j;
for (i=0; i<x; ++i)
field->t[i] = &field->storage[i*y];
if (sourceX)
{
double scaleFactorX = (double)sourceX/x;
double scaleFactorY = (double)sourceY/y;
for (i=0; i<x; ++i)
for (j=0; j<y; ++j)
field->t[i][j] = temp.t[(int)(i*scaleFactorX)][(int)(j*scaleFactorY)];
deleteField(&temp);
}
else memset(field->storage, 0, sizeof(double)*x*y);
}

void initField(TemperatureField *field)
{
int i, j;
for (i=0; i<field->x; ++i)
for (j=0; j<field->y; ++j)
field->t[i][j] = 20.0f;
}

void refreshField(TemperatureField *field, int initX, int initY, int thisX, int this
{
int j;
for (j=allY*3/10; j<allY*7/10; ++j)
if (legal(-initX, thisX)&&legal(j-initY, thisY))
field->t[-initX][j-initY] = 100.0f;
}

TemperatureField* myClone(TemperatureField *field, int X, int Y)
{
int i, j;
TemperatureField *ret = malloc(sizeof(TemperatureField));
ret->x = X;
ret->y = Y;
ret->storage = malloc(sizeof(double)*ret->x*ret->y);
ret->t = malloc(sizeof(double*)*ret->x);
for (i=0; i<ret->x; ++i)
ret->t[i] = &ret->storage[i*ret->y];
for (i=0; i<X; ++i)
for (j=0; j<Y; ++j)
ret->t[i][j] = field->t[i][j];
return ret;
}

void deleteField(TemperatureField *field)
{
free(field->t);
free(field->storage);
//free(field);
}

#endif
```

The sequential version.

```
#include "const.h"
#include "models.h"
#include "display.h"

#define legal(x, n) ( (x)>=0 && (x)<(n) )

int iteration,x,y;
TemperatureField *field;
TemperatureField *tempField, *swapField;

int dx[4] = {0, -1, 0, 1};
int dy[4] = {1, 0, -1, 0};

void temperature_iterate(TemperatureField *field, int x)
{
int i, j, d;
for (i=0; i<field->x; ++i)
for (j=0; j<field->y; ++j)
{
int cnt = 0;
tempField->t[i][j] = 0;
for (d=0; d<4; ++d)
if ( legal(i+dx[d], field->x) && legal(j+dy[d], field->y) )
{
tempField->t[i][j] += field->t[i+dx[d]][j+dy[d]];
++cnt;
}
tempField->t[i][j] /= cnt;
}
for (i=0;i<7*x/10;i++)
{
if(3*x/10 <i)
tempField->t[0][i] = 100.0f;
}
}

int main(int argc, char **argv)
{
if (argc<4)
{
```

```
printf("Usage:␣%s␣x␣y␣iteration\n", argv[0]);
}
sscanf(argv[1], "%d", &x);
sscanf(argv[2], "%d", &y);
sscanf(argv[3], "%d", &iteration);

field = malloc(sizeof(TemperatureField));
tempField = malloc(sizeof(TemperatureField));
newField(field, x, y,0,0);
newField(tempField, x, y,0,0);
initField(field);
XWindow_Init(field);

int iter;
for (iter=0; iter<iteration; iter++)
{
temperature_iterate(field, x);
swapField = field;
field = tempField;
tempField = swapField;
if(iter % 100 == 0) XRedraw(field);
}
return 0;
}
```

The pthread version.

```
#include "const.h"
#include "models.h"
#include "display.h"
#include <pthread.h>
#include <stdio.h>

#define legal(x, n) ( (x)>=0 && (x)<(n) )
#define start_time clock_gettime(CLOCK_MONOTONIC, &start);
#define end_time clock_gettime(CLOCK_MONOTONIC, &finish);
#define time_elapsed_ns (long long)(finish.tv_sec-start.tv_sec)*1000000000 + finish
#define time_elapsed_s (double)(finish.tv_sec-start.tv_sec) + (double)(finish.tv_nse
#define NOT_FIRE_PLACE i

int iteration, threads;

TemperatureField *field;
TemperatureField *tempField, *swapField;

pthread_t *threadPool;
pthread_mutex_t *subThreadWakeUp, *subThreadFinished;
int *threadID, terminate;

double *error;
double  EPSILON;


int dx[4] = {0, -1, 0, 1};
int dy[4] = {1, 0, -1, 0};

int x, y, iter_cnt;

int min(int x, int y){ if (x<y) return x; return y; }

void* iterateLine(void* data)
{
int threadID = *((int*)data);
while (1)
{
/*Lock the thread calculating now, then set the size, start and end */
pthread_mutex_lock(&subThreadWakeUp[threadID]);
if (terminate) break;
int blockSize = field->x/threads + !!(field->x%threads);
int lineStart = blockSize * threadID;
int lineEnd = min(blockSize*(threadID+1), field->x);
error[threadID]=0;

int i, j, d;
for (i=lineStart; i<lineEnd; ++i)
for (j=0; j<field->y; ++j)
{
tempField->t[i][j] = 0;
for (d=0; d<4; ++d)
if ( legal(i+dx[d], field->x) && legal(j+dy[d], field->y) )
tempField->t[i][j] += field->t[i+dx[d]][j+dy[d]];
else
tempField->t[i][j] += ROOM_TEMP;
tempField->t[i][j] /= 4;
if (NOT_FIRE_PLACE)
error[threadID] += fabs(tempField->t[i][j] - field->t[i][j]);
}
/* add the thread finished job to finished*/
pthread_mutex_unlock(&subThreadFinished[threadID]);
}
pthread_exit(NULL);
}

double temperature_iterate()
{
++iter_cnt; //Just a counter hold the iteration number.
refreshField(field, 0, 0, field->x, field->y, field->x, field->y);
int i;

/* unlock threads in WakeUp and lock threads in Finish*/
for (i=0; i<threads; ++i)
pthread_mutex_unlock(&subThreadWakeUp[i]);
for (i=0; i<threads; ++i)
pthread_mutex_lock(&subThreadFinished[i]);

double sumError = 0;
for (i=0; i<threads; ++i)
sumError += error[i];

return sumError;
}
```

```c

int main(int argc, char **argv)
{
struct timespec start, finish;
start_time

/*Reading parameter*/
if (argc<5)
{
printf("Usage:␣%s␣x␣y␣iteration␣INCREMENT_TIME
,␣INCREMENT␣threads␣EPSILON\n", argv[0]);
}
sscanf(argv[1], "%d", &x);
sscanf(argv[2], "%d", &y);
sscanf(argv[3], "%d", &iteration);
sscanf(argv[4], "%d", &threads);
sscanf(argv[5], "%lf", &EPSILON);

field = malloc(sizeof(TemperatureField));
tempField = malloc(sizeof(TemperatureField));
threadPool = malloc(sizeof(pthread_t)*threads);
subThreadWakeUp = malloc(sizeof(pthread_mutex_t)*threads);
subThreadFinished = malloc(sizeof(pthread_mutex_t)*threads);
threadID = malloc(sizeof(int)*threads);
error = malloc(sizeof(double)*threads);
terminate = 0;
field->x = y;
field->y = x;



/*Initial mutex and lock up*/
int i;
for (i=0; i<threads; ++i)
{
pthread_mutex_init(&subThreadWakeUp[i], NULL);
pthread_mutex_init(&subThreadFinished[i], NULL);
pthread_mutex_lock(&subThreadWakeUp[i]);
pthread_mutex_lock(&subThreadFinished[i]);
threadID[i] = i;
pthread_create(&threadPool[i], NULL, iterateLine, &threadID[i]);
}

int iter;
newField(field, x, x, 0, 0);
newField(tempField, x, x, 0, 0);
initField(field);
XWindow_Init(field);

/* Main iteration, lines is computed parallel*/
for (iter=0; iter<iteration; iter++)
{
double error = temperature_iterate();
if (error<EPSILON)
{
```

```
printf("Finished.␣iteration=%d,␣error=%lf\n", iter, error);
break;
}
swapField = field;
field = tempField;
tempField = swapField;
if(iter % 100 == 0) XRedraw(field);
}

/*Delete field and unlock mutex*/
deleteField(field);
deleteField(tempField);
free(threadPool);
for (i=0; i<threads; ++i)
{
terminate = 1;
pthread_mutex_unlock(&subThreadWakeUp[i]);
}

/*Print result and exit multi-threads*/
printf("Finished␣in␣%d␣iterations.\n", iter_cnt);
end_time;
printf("%lf\n", time_elapsed_s);

sleep(30);
pthread_exit(NULL);
return 0;
}
```

The mpi version.

```
#include "models.h"
#include "display.h"
#include <mpi.h>

#define legal(x, n) ( (x)>=0 && (x)<(n) )
#define start_time clock_gettime(CLOCK_MONOTONIC, &start);
#define end_time clock_gettime(CLOCK_MONOTONIC, &finish);
#define time_elapsed_ns (long long)(finish.tv_sec-start.tv_sec)*1000000000 + finish
#define time_elapsed_s (double)(finish.tv_sec-start.tv_sec) + (double)(finish.tv_nse
#define NOT_FIRE_PLACE i;

int job;

int dx[4] = {0, -1, 0, 1};
int dy[4] = {1, 0, -1, 0};

int iteration,x,y,iter_cnt;
TemperatureField *field;
TemperatureField *tempField, *swapField;

void temperature_iterate(TemperatureField *field, int x)
{
++iter_cnt; //Just a counter hold the iteration number.
int i, j, d;
for (i=0; i<field->x; ++i)
for (j=0; j<field->y; ++j)
{
int cnt = 0;
tempField->t[i][j] = 0;
for (d=0; d<4; ++d)
if ( legal(i+dx[d], field->x) && legal(j+dy[d], field->y) )
{
tempField->t[i][j] += field->t[i+dx[d]][j+dy[d]];
++cnt;
}
tempField->t[i][j] /= cnt;
}
for (i=0;i<7*x/10;i++)
{
if(3*x/10 <i)
tempField->t[0][i] = 100.0f;
}
}

int main(int argc, char **argv){

XInitThreads();

struct timespec start, finish;
start_time

int i;
int size, rank;

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if (argc<4) {
printf("Usage:␣%s␣x␣iteration\n", argv[0]);
}
sscanf(argv[1], "%d", &x);
sscanf(argv[1], "%d", &y);
sscanf(argv[3], "%d", &iteration);

field = malloc(sizeof(TemperatureField));
tempField = malloc(sizeof(TemperatureField));
newField(field, x, x, 0, 0);
newField(tempField, x, x, 0, 0);
initField(field);

if (rank == 0) {
XWindow_Init(field);
}

job = x / size;
if (x % size != 0) job++;

int startx = rank * job;

int iter;
for (iter = 0; iter < iteration; iter++) {
temperature_iterate(field, startx);
MPI_Allgather(&(tempField->t[startx][0]), job*field->y, MPI_FLOAT,
 &(field->t[0][0]), job*field->y, MPI_FLOAT, MPI_COMM_WORLD);

if (rank == 0) {
for(i = x * 0.3; i < x * 0.7; i++)
field->t[0][i] = FIRE_TEMP;
if(iter % 100 == 0) XRedraw(field);
}
}

if (rank == 0) {
/*Print result and exit multi-threads*/
printf("Finished␣in␣%d␣iterations.\n", iter_cnt);
end_time;
printf("%lf\n", time_elapsed_s);
}

MPI_Finalize();
return 0;
}
```

References

[1] The Latex Template used for assignment is cite from overleaf. *https : //www.overleaf.com/latex/templates/ece − 100 − template/pjrrfybfggqt*

[2] The source code for sequential program is spread by the instructor from the site. *http : //www.cs.nthu.edu.tw/ ychung/homework/para*$_p$*rogramming.htm*

[3] Part of codes are provided by tutor on tutorial section cited from GitHub.*https : //github.com/cjf00000/Heat − Distribution*

[4] Temperature color standard *https : //wenku.baidu.com/view/27f15d48cf84b9d528ea7ae8.html*