# The N-Body and Heat Distribution Problems

Homework Three

***William Martin***
Parallel and Distributed Algorithms
CSC673
Spring 2003
Due Date: April 7, 2003

**To:** Dr. Jeff Mcgough
**From:** William Martin
**Re:** Homework #3, 4/7/03

---

## Introduction

The goal of this assignment is to develop sequential and parallel systems for two problems assigned to the "Parallel and Distributed Algorithms" class. The task for this assignment consists of a pseudo real-time Astronomy simulation of the gravitational effects on an N-body system and a Heat distribution simulation problem.

1. N-Body Problem (text): 4-20
2. Heat distribution (text): 6-14

How to compile:

> make nbody: To make the nbody executable
> make heat: To make the heat distribution executable
> make all: to make nbody and heat

How to run:

mpirun –c n nbody or heat
where n is the number of processors.

# *Problem 6-14*

## *Problem Description:*

Figure 6.19 (in the book, p192) shows a room that has four walls and a fireplace. The temperature of the wall is 20 degrees, centigrade and the temperature of the fireplace is 100 degrees centigrade. Write a parallel program using **Jacobi iteration** to compute the temperature inside the room and plot (preferably in color) temperature contours at 10 degrees Centigrade intervals using Xlib calls or similar graphics calls as available on your system. Instrument the code so that the elapsed time is displayed. (the programming assignment is convenient after a Mandelbrot assignment because it can use the same graphics calls).

### *More Details*

The heat distribution problem described in the textbook, on page 179, discusses this as a local synchronization problem. A local synchronization problem is one that consists of synchronizing itself with only a subset of the running processes, whereas a global synchronization problem is one that consists of synchronization with all processes.

Without much imagination, one could foresee that the processes involved in a local synchronization problem of heat distribution would involve only the pixel's own neighborhood; propagating heat from an area of higher temperature to lower. To be more specific, it would involve only its nearest neighbors.

## *Sequential Implementation with XdrawPoint and PutPixel*

Just a brief note on the sequential/serial implementation; it is **extremely** slow. I tried the simple Xlib call, **XdrawPoint**() and Brian Stone's *fast* **PutPixel**() routine and saw no real difference. It was basically a shot in the dark, because I had implemented the Xdrawpoint version first and noted the speed was <u>unacceptable</u>, even at 200x200. I thought that using Brian Stone's method would be visibly faster, and spent a few minutes filling in the changes needed. It was not time well spent.

I would also like to add that there is a likelihood of my sequential implementation not being optimal. I followed the textbook's example for sequential heat distribution, which is probably more likely a "user-readable" version vs. an optimized version.

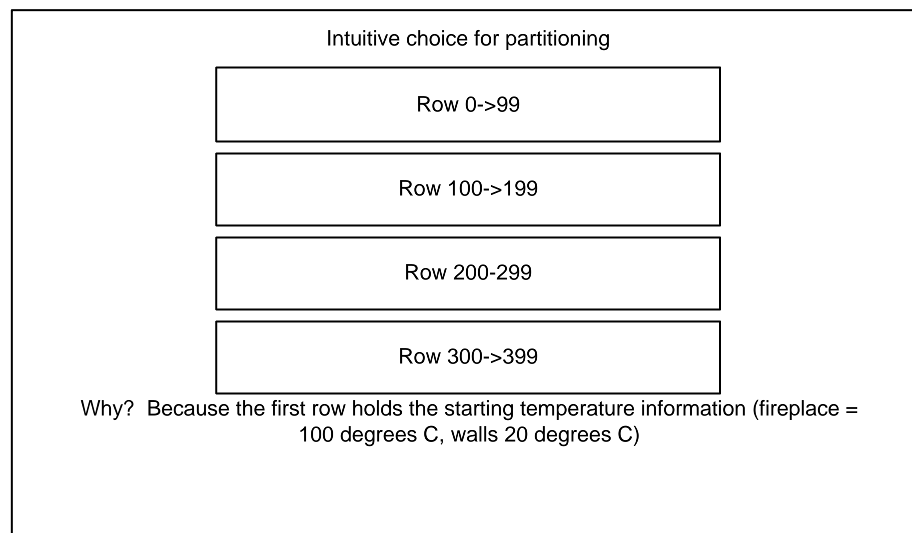
## *Parallel Implementation*

From the text and Hewlett Packard's HPC site, I've read that Jacobi's iterative method is slow to converge. Why use it then? Hewlett Packard states that it is an excellent method for teaching parallelization, because it is simple to implement. HP also

goes on to recommend optimizations, such as reducing inter-node communication and "shadow edges", or 'ghost points' as described in class (lecture 21, 3/14/2003).

***Partitioning***

As described in the book, we have the choice of either a block or strip partitioning for dividing up data to processors. The block partition is better if we have a small startup time. The strip partitioning is better if we have a large startup time. Why is this? Because the amount of communication that exists between partitions in the block scheme is much greater than the strip partitioning. For MPI, it is best to use "strip partitioning", and thus I have used that for this project.




I've also decided to implement the strips into rows, instead of columns. The textbook presents the fact that C is row-major order and thus is the logical choice. I believe that there is another, more valid, reason which is problem-description specific: the fireplace and walls are located at the top of the room.

In order to prevent "locks" when sending data from one node to another, I implemented the even/odd arrangement discussed in the book – which is, the even node will send first, while the odd nodes receive. And, then the even will receive, while the odd will send.

***Global Termination***

And, for global termination, I have each node send up "*converged*" or "*not converged*" to the master process. If it receives all "converged", then it sends back a message to state that its time to quit and send all the rows you currently have. It then proceeds to draw that data and exits immediately. The choice to leave out another method of exiting was decided based upon the request that the professor not have to

spend his/her life figuring out if he/she should manually exit, or should wait on the master to exit.

## *Problem 4-20*

**4-20:** Write a sequential program and a parallel program to simulate an astronomical N-body system, but in two-dimensions. The bodies are initially at rest. Their initial positions and masses are to be selected randomly. Display the movement of the bodies using the graphical routines used for the Mandelbrot program found in http://www.cs.uncc.edu/par_prog, or otherwise, showing each body in a color and size to indicate the mass.

The problem of simulating an N-body calculation is an interesting and challenging problem. The major idea behind this algorithm is to simulate the gravitational effects on a body in space, given one or more heavenly bodies. This involves computing the impact on every body in the given 'universe'. Realistically, there are many theoretical aspects that are being left out such as black holes and dark matter. For the purpose of this assignment, we will not dwell on the collision Boltzmann equation or consult Stephen Hawking or do any further research except to attempt to author a sequential and a parallelized solution to the given problem.

With that in mind we, as an individual, have a choice as to whether to implement this problem using the straightforward code in the book, or using another method called the "Barnes-Hutt Algorithm". There are many variations of the Barnes-Hutt Algorithm, which is a great algorithm because of the time complexity (on the order of log n). It is also recursive, which lends itself as a great solution to implement in a divide and conquer fashion. In dealing with large systems, it is best to use a faster algorithm. The N-body solution in the book is O(N^2); unrealistic for larger systems.

I have chosen to use the simple choice of the universal gravitational force equation.

$$F = \frac{GmM}{r^2}$$

One thing a person has to acknowledge is that the force equation is calculated for each dimension that the programmer intends to design with. In the simple case of my program, I have chosen to implement the 2-dimensional version, and thus only the x and y coordinates are needed.

# *Scaling*

During a discussion in class, we mentioned the fact that the universe would collapse upon itself if we used the gravitational constant, g, instead of G. g= 9.8 m/s, whereas G =6.711 x 10-11. It is true, if the rest of the universe is modeled realistically, that we will see a very quick destruction of most of the objects in our Universe. To prove that all of the objects would be destroyed is another interesting aspect of this assignment, of which I will not have time to investigate and research.

My original (and consequently final) implementation does not collapse upon itself. The reason is because I've scaled the other *factors* involved in the calculation. I limited the masses to be <=2000 and the delta t (our sampling period, and also the elapsed time between "epochs") to be 0.5. This produced nice results; as long as I used a "molasses-factor", i.e. "sleep" for a certain amount of time before making the next calculation of the epoch.

# *Parallelization*

The implementation of this requires a barrier for parallelization because we have a need for synchronization after each object's data (x,y,vx,vy) is calculated. The slave nodes must have the current location of the other nodes in order to calculate the effect of its mass. The barrier description is briefly demonstrated in Figure 1.0. Essentially, we will initialize the data (this could be done at the Master or individually, and more complex, by the slaves) and then send out the initialized data to all nodes. All nodes will go through one iteration of calculating it's set of nodes and then send that data back to the master.

# *Termination*

The nbody program will exit on it's own. It has been clocked to give at least some objects the ability to leave the "universe", but maybe a few stragglers will stick around to give the professor some trails to look at, and some visible bodies interacting.

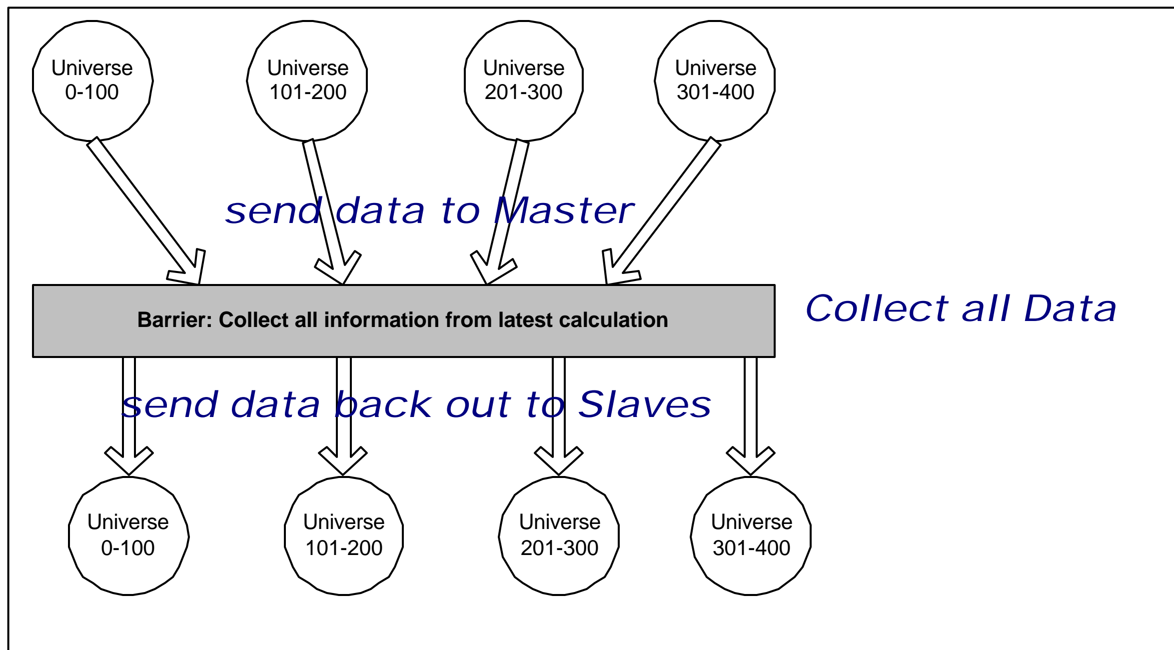



***Figure 1.0: Simple Barrier Description***

## Nbody Source Code (Parallel version)

```
/***********************************************************************
**      Program         :   nbody
**      Written by      :   William Martin
**      Date            :   February 27, 2003
**      Purpose         :   To simulate an N-body gravitational system
**                          problem.
**      how-to-call     :   nbody n, where n is the number of iterations
**      The N-body problem is discussed in Chapter 4, page 126
**
***********************************************************************/
#include <X11/Xlib.h> /* Every Xlib program must include this*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/types.h>
#include <unistd.h>
```

```
#define NIL (0)
#include <mpi.h>
#define GBL_WIDTH 800.0
#define GBL_HEIGHT 600.0
//////////////////////////////////////////////////////////
// NOTE: CHANGE THIS LINE TO HAVE MORE OR LESS OBJECTS
/////////////////////////////////////////////////////////
#define GBL_NUMOBJECTS  150
/////////////////////////////////////////////////////////
// change this line to have a delay time in drawing
////////////////////////////////////////////////////////
#define DELAYTIME       12000
////////////////////////////////////////////////////////
// how many "time periods"
///////////////////////////////////////////////////////
#define NUMBER_OF_EPOCHS 850

#define WORKTAG 25       /* MY "SPECIAL" TAG */

/*
struct twodpos {
        double x;
        double y;
        double vx;define SDEBUG 0
        double vy;
        double mass;
};
*/
//////////////////////////////////////////////////////////////
// refernce the above structure to figure
// out why these variables are set so.
// They reference the single-dimensional object, retpts
// retpts2.
/////////////////////////////////////////////////////////////
#define xpos 0
#define ypos 1
#define vxpos 2
#define vypos 3
#define masspos 4

#define DEBUG 1

double object[GBL_NUMOBJECTS*5];
double G_FORCE;

/////////////////////////////////////////////////////////////
// function pre-declares
/////////////////////////////////////////////////////////////
double Force_routine(int current_body, char component, double object1[]);
void CreateUniverse(int numbodies);
void InitGraphics();
void RedrawScreen();

/* global variables */
Display* dpy;
Window w;
GC gc;

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

        int             tmax;
        /* loop counters */
        int             t, i,x,j;
        /* MPI Variables */
        int             myrank = 0;    /* who am I? */
        int             size = 0;
        int             totalsize;
        int             leftovers=0;
```

```
        MPI_Status      status;
        double retpts[GBL_NUMOBJECTS*5];
        double retpts2[GBL_NUMOBJECTS*5];
        double          F = 0.0;
        double          dt = 0.2;       /* change in time=3 */


        G_FORCE = 5.9;

        tmax = NUMBER_OF_EPOCHS;

        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        MPI_Comm_rank(MPI_COMM_WORLD, &myrank);/* get the rank */

        totalsize = (GBL_NUMOBJECTS)/(size-1);
        leftovers =GBL_NUMOBJECTS % (size-1);

        if(myrank==0) {         /* we are the master */
                InitGraphics();
                CreateUniverse(GBL_NUMOBJECTS);

                for(t=0;t<tmax;t++) {
                        for(x = 1;x < size;x++) {
                                MPI_Send(object,GBL_NUMOBJECTS*5, MPI_DOUBLE, x,
WORKTAG,MPI_COMM_WORLD);
                        }
                        for(x = 1;x < size;x++) {
                                MPI_Recv(&retpts2, GBL_NUMOBJECTS*5, MPI_DOUBLE,
MPI_ANY_SOURCE, WORKTAG,MPI_COMM_WORLD, &status);
                                for(i=0;i<totalsize*5;i++){
                                        object[(((status.MPI_SOURCE-1)* totalsize * 5) +
i)] = retpts2[i];
                                }

                        }
                        /* if we have leftovers, let the master handle them. */
                        if(leftovers>0)  {

                                for(i=(GBL_NUMOBJECTS-
leftovers)*5;i<GBL_NUMOBJECTS*5;i+=5) {
                                       F = Force_routine(i, 'x', object);      /*compute
force on ith body */
                                        object[vxpos + i] = object[vxpos + i] + F *
(double)dt/(double)object[masspos + i];
                                        F = Force_routine(i, 'y', object);
                                        object[vypos + i] = object[vypos + i] + F *
(double)dt/(double)object[masspos + i];
                                        object[xpos + i] = object[xpos + i] +
object[vxpos + i] * dt;
                                        object[ypos + i] = object[ypos + i] +
object[vypos + i] * dt;


                                }
                        }

                        RedrawScreen();
                }

               printf("Done...now sleeping 1 second.  Do not close window or press ctrl-
c in the shell window. \n");
                sleep(1);

        }
        else { /* slave n */

                for(t=0;t<tmax;t++) {
```

```
                        for(i=((myrank-1)*totalsize*5);i<((myrank-1)*totalsize*5) +
(totalsize*5);i+=5) {
                        MPI_Recv(&object, GBL_NUMOBJECTS*5, MPI_DOUBLE, 0, WORKTAG,
MPI_COMM_WORLD, &status);
                                F = Force_routine(i, 'x', object);      /*compute force
on ith body */
                                object[vxpos + i] = object[vxpos + i] + F *
(double)dt/(double)object[masspos + i];
                                F = Force_routine(i, 'y', object);
                                object[vypos + i] = object[vypos + i] + F *
(double)dt/(double)object[masspos + i];
                                object[xpos + i] = object[xpos + i] + object[vxpos + i] *
dt;
                                object[ypos + i] = object[ypos + i] + object[vypos + i] *
dt;

                        }
                        for(j=0;j<totalsize*5;j++) {
                                retpts[j] = object[(((myrank-1)*totalsize*5)+j)];
                        }
                        MPI_Send(retpts, GBL_NUMOBJECTS*5, MPI_DOUBLE, 0, WORKTAG,
MPI_COMM_WORLD);
                }

        }
        return 0;
}


/*****************************************************************************
**              FUNCTION:       Force_routine
**              PURPOSE:        Calculate effect of other n-1 bodies on body.
*****************************************************************************/
double Force_routine(int current_body, char component, double object1[]) {
        double result=0.0;
        int i=0;
        double second=0.0;
        double r=0;
        //printf("Current body =%d \n",current_body);
        ////////////////////////////////////////////
        //  Enumerate all of the bodies
        ////////////////////////////////////////////
        for(i=0;i<GBL_NUMOBJECTS*5;i+=5) {
                if(i!=current_body) {

                        r = sqrt(pow(object1[xpos+i]-object1[xpos+current_body],2.0) +
                            pow(object1[ypos + i]-object1[ypos + current_body],2.0));


                        if(r!=0) {
                                if(component=='x') {
                                        second = (object1[xpos+i]-
object1[xpos+current_body])/r;
                                }
                                else {

                                        second = (object1[ypos+i]-object1[ypos +
current_body])/r;
                                }

                                result += (((object1[masspos+current_body] *
object1[masspos+i]) * G_FORCE)/(pow(r,2.0))*second);
                        }
                }
        }
        result=result/(double)GBL_NUMOBJECTS;

        return result;
}
```

```
**      FUNCTION:       CreateUniverse
**      PURPOSE :       To initialize the N-bodies in space.
*****************************************************************************/
void CreateUniverse(int numbodies) {
        int i;
        int v=getpid();
        srand(v);

        for(i = 0;i < GBL_NUMOBJECTS*5;i+=5) {
                object[xpos+i]= 1.0 + (double)((double) GBL_WIDTH
*rand()/(double)(RAND_MAX+1.0));
                object[ypos+i]= 1.0 + (double)((double) GBL_HEIGHT
*rand()/(double)(RAND_MAX+1.0));
                object[masspos+i] =1.0 +(double)(16000.0 *
rand()/(double)(RAND_MAX+1.0));
                object[vxpos+i] = 0.0;
                object[vypos+i] = 0.0;

        }

}
/*****************************************************************************
**      FUNCTION:       InitGraphics
**      PURPOSE :       Initializes graphics
*****************************************************************************/
void InitGraphics() {
        /*create a display object */
        /* create a Graphics context object */
        /* GC gc; */
        /* reserve two colors for use */
        Display* newdpy = XOpenDisplay(NIL);
        int blackColor = BlackPixel(newdpy, DefaultScreen(newdpy));
        int whiteColor = WhitePixel(newdpy, DefaultScreen(newdpy));
        dpy = newdpy;
        /* create a simple window, h200 x w100, black bg */
        w = XCreateSimpleWindow(dpy, DefaultRootWindow(dpy), 0, 0,
        GBL_WIDTH, GBL_HEIGHT, 0, blackColor, blackColor);
        XSelectInput(dpy, w, StructureNotifyMask);
        XMapWindow(dpy, w);
        /* initialize GC */
        gc = XCreateGC(dpy, w, 0, NIL);
        /* set the forecolor */
        XSetForeground(dpy, gc, whiteColor);
        for(;;) {
                XEvent e;
                XNextEvent(dpy, &e);
                if (e.type == MapNotify)
                        break;
                }
        XFlush(dpy);
}

/*****************************************************************************
**      FUNCTION:       RedrawScreen
**      PURPOSE :       redraws the screen and objects new positions
*****************************************************************************/
void RedrawScreen() {

        int color=5000;
        int x=0;
        long i=0;
        double dummy=0.0;

        XFlush(dpy);

        /*********************************************************************
        ** this implements a "delay".  Sleep fails to redraw thereafter
        *********************************************************************/
        for(i=0;i<DELAYTIME;i++) //32765
```

```
        XSetForeground(dpy,gc,color);
                 dummy = pow(dummy,2.0);
        for(x=0;x<GBL_NUMOBJECTS*5;x+=5) {
                if(object[masspos+x] < 2100.0) {        /* blue */
                        XSetForeground(dpy,gc,0x0000ff);
                        XDrawPoint(dpy,w,gc,(int)object[xpos+x]+1,(int)object[ypos+x]);
                        XDrawPoint(dpy,w,gc,(int)object[xpos+x]-1,(int)object[ypos+x]);
                        XDrawPoint(dpy,w,gc,(int)object[xpos+x]-1,(int)object[ypos+x]+1);
                        XDrawPoint(dpy,w,gc,(int)object[xpos+x]+1,(int)object[ypos+x]-1);
                }
                else if(object[masspos+x] <5600.0) {    /*white */
                        XSetForeground(dpy,gc,0xFFFFFF);
                        XDrawPoint(dpy,w,gc,(int)object[xpos+x],(int)object[ypos+x]-1);
                        XDrawPoint(dpy,w,gc,(int)object[xpos+x],(int)object[ypos+x]+1);
                }
                else {  /* green */
    XSetForeground(dpy,gc,0xFF0F00);


XDrawPoint(dpy,w,gc,(int)object[xpos+x],(int)object[ypos+x]-1);
                        XDrawPoint(dpy,w,gc,(int)object[xpos+x],(int)object[ypos+x]+1);
                        XDrawPoint(dpy,w,gc,(int)object[xpos+x]-1,(int)object[ypos+x]);
                        XDrawPoint(dpy,w,gc,(int)object[xpos+x]+1,(int)object[ypos+x]);
                        XDrawPoint(dpy,w,gc,(int)object[xpos+x]+1,(int)object[ypos+x]-1);
}
XDrawPoint(dpy,w,gc,(int)object[xpos+x],(int)object[ypos+x]);

        }
        XFlush(dpy);
}

        /***************************************
        **    E N D  O F  P R O G R A M     **
        ***************************************/
```

## Heat Distribution Source Code

```
/*********************************************************************************
**      Name        :    Heat Distribution Problem
**      Filename    :    heatdistro.c
**      Written by  :    William Martin
**      Purpose     :    To implement the heat distribution problem, 6-14
**      Problem
**      Statement   :  Figure 6.19 shows a room that has four walls and a
**                     fireplace.  The temperature of the wall is
**                     20 degrees centigrade, and the temperature is 100 degrees
**                     centigrade.  Write a parallel program using jacobi
**                     iteration to compute the temperature inside the room and
**                     plot (preferably in color) temperature contours at 10
**                     degree intervals using Xlib calls or similar graphics
**                     calls as available on your system. Instrument the code
**                     so that the elapsed time is displayed.  (this programming
**                     assignment is convenient after a Mandelbrot assignment
**                     because it can use the same graphics calls.
**
**     Note:           This DOES NOT ACCOUNT FOR "NON-INTEGER" splits!!!
**                     **** **** *** ******* ***  ***********  ******
********************************************************************************
*********************************************************************************/
#include <stdio.h>
#include <time.h>
```

```
#include <X11/Xlib.h>
#include <mpi.h>
#include <math.h>
#include <unistd.h>

/* A name for the void pointer*/
#define NIL (0)
#define WORKTAG         21
#define QUITTAG         86      /*       Special tag to say       */
                                /* "The Master wishes you to quit"*/
#define ROOMWIDTH       400
#define ROOMHEIGHT      400
#define FIREPLACETEMP   100
#define WALLTEMP        20
#define DEBUG 1

enum COLORS {BLACK=0X000000,RED=0x00f000,BLUE=0x0000ff, PURPLE=0x00f0Ff, PINK=0xf1f00f,
GREEN=0x000f00,
                LIGHTGREEN=0xff0f00,YELLOW=0x00ff00};
enum BOOLEAN {FALSE=0,TRUE = 1};

/////////////////////////////////////////////
// Routine List
//-------------------------------------------
// InitGraphics()
// DisplayElapsedTime()
// DoDistribution()
// PartitionSize()
// CalculateSlaveValues()
/////////////////////////////////////////////
void InitGraphics();
void InitializeWorkspace();
void DoDistribution(int numprocs, int currentrank);
void DrawGraphics();
void PartitionData(int numprocs);
void CalculateSlaveValues();

int myrank;
int size;
int global_term;
/////////////////////////////////////////////
// Declare basic graphics elements
/////////////////////////////////////////////
Display* dpy;
Window w;
GC gc;

struct timeval starttime;


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

        //////////////////////////////////////////////////////
        //      initialize MPI variables
        //////////////////////////////////////////////////////
        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
        global_term=FALSE;

        if(myrank == 0) { /* master; Role of the 'painter' */
                gettimeofday(&starttime,(struct timezone *) 0);
                InitGraphics();
                PartitionData(size);
        }
        else { /* do the slave work */
                DoDistribution(size,myrank);
        }
        MPI_Finalize();
        return(0);
```

```
}

//////////////////////////////////////////////////////////////////////////
//  Function:    InitGraphics
//  Purpose:     To initialize the graphics context and set up
//               colors.
/////////////////////////////////////////////////////////////////////////
void InitGraphics() {
        /*create a display object */
        /* create a Graphics context object */
        /* GC gc; */
        /* reserve two colors for use */

        Display* newdpy = XOpenDisplay(NIL);

        int blackColor = BlackPixel(newdpy, DefaultScreen(newdpy));
        int whiteColor = WhitePixel(newdpy, DefaultScreen(newdpy));


        dpy = newdpy;

        /* create a simple window, h200 x w100, black bg */
        w = XCreateSimpleWindow(dpy, DefaultRootWindow(dpy), 0, 0,
        ROOMWIDTH, ROOMHEIGHT, 0, blackColor, blackColor);

        XSelectInput(dpy, w, StructureNotifyMask);
        XMapWindow(dpy, w);

        /* initialize GC */
        gc = XCreateGC(dpy, w, 0, NIL);

        /* set the forecolor */
        XSetForeground(dpy, gc, whiteColor);

        for(;;) {
                XEvent e;
                XNextEvent(dpy, &e);
                if (e.type == MapNotify)
                        break;
                if(global_term==TRUE)
                        return;
                }

        XFlush(dpy);
        XSetForeground(dpy, gc, RED);
        XDrawPoint(dpy,w,gc,50,50);
}


//////////////////////////////////////////////////////////////////////
//      Function : DoDistribution
//      Purpose  : This is responsible for doing the heat
//                 distribution
//
/////////////////////////////////////////////////////////////////////
void DoDistribution(int numprocs, int currentrank) {

        int iteration=0;
        int limit = 5355;
        int terminate = FALSE;
        int gterminate = FALSE;
        int i,j;
        MPI_Status status;
        MPI_Request req;

        unsigned long startingpoint = 0;
        long rowsize = ROOMHEIGHT/(numprocs-1);

        double habove[ROOMWIDTH]={20.0};           /* represents ghost points*/
```

```
        double gabove[ROOMWIDTH]={20.0};            /* represents ghost points*/
        double gbelow[ROOMWIDTH]={20.0};            /* represents ghost points*/

        double hbelow[ROOMWIDTH]={20.0};            /* represents ghost points*/

        double h[numprocs][ROOMWIDTH][rowsize];          /* double*/
        int converged[ROOMWIDTH][rowsize];      /* int */

        double oldvalues[ROOMWIDTH][rowsize];   /* double*/

        double g[numprocs-1][ROOMWIDTH][rowsize];             /* double*/

        //////////////////////////////////////////////////
        // Initialize values for running
        /////////////////////////////////////////////////
        for(i = 0;i < ROOMWIDTH;i++){
                for(j = 0;j<rowsize;j++) {
                        converged[i][j] = 0;
                        oldvalues[i][j] = -50.0;
                        h[currentrank][i][j]=20.0;
                        g[currentrank][i][j]=20.0;
                }
        }


        while(terminate==FALSE) {
                ///////////////////////////////////////////
                // Set up fireplace, and walls
                //////////////////////////////////////////
                if(currentrank==1) {
                        for(i = (ROOMWIDTH * (1.0/3.0));i < (ROOMWIDTH * (1.0/3.0) ) *
2;i++) {
                                h[currentrank][i][0]=FIREPLACETEMP;
                                h[currentrank][i][0]=FIREPLACETEMP;
                        }

                        for(i = 0;i < (ROOMWIDTH * (1.0/3.0));i++) {
                                h[currentrank][i][0]=WALLTEMP;
                                h[currentrank][i][0]=WALLTEMP;
                        }
                        for(i = ROOMWIDTH*(1.0/3.0) * 2;i <ROOMWIDTH;i++) {
                                h[currentrank][i][0]=WALLTEMP;
                                h[currentrank][i][0]=WALLTEMP;
                        }
                }
                //////////////////////////////////////////////////////////////
                //  Modified Jacobi Iterative method.
                //////////////////////////////////////////////////////////////
                for(i = 1;i < ROOMWIDTH-1 ;i++) {
                        for(j = 0;j < rowsize;j++) {
                                if(j==0 && currentrank==1)
                                        g[currentrank][i][j] = 0.25 * (h[currentrank][i-
1][j] + h[currentrank][i+1][j] + h[currentrank][i][j+1]);
                                else if(j==0 && currentrank>1) {
                                        g[currentrank][i][j] = 0.25 * (h[currentrank][i-
1][j] + h[currentrank][i+1][j] + hbelow[i] + h[currentrank][i][j+1]);
                                }
                                else if(j==rowsize-1 && currentrank<(numprocs-1)) {
                                        g[currentrank][i][j] = 0.25 * (h[currentrank][i-
1][j] + h[currentrank][i+1][j] + h[currentrank][i][j-1] + habove[i]);
                                }
                                else    {
                                        if(j!=rowsize-1)
                                                g[currentrank][i][j] = 0.25 *
(h[currentrank][i-1][j] + h[currentrank][i+1][j] + h[currentrank][i][j-1] +
h[currentrank][i][j+1]);
                                }
                                //////////////////////////////////////////////////
                                // While we're looking at the node, has it changed
                                // much from it's previous value?
```

```
                        ////////////////////////////////////////////////////
                        if(abs(g[currentrank][i][j]-oldvalues[i][j])< 0.0001) {
                        // NOTE: CONVERGENCE CHECKING HERE.
                                converged[i][j] = 1;
                        }
                        oldvalues[i][j] = g[currentrank][i][j];
                }
        }

        ///////////////////////////////////////////////////////////////
        //      save off computed values
        ///////////////////////////////////////////////////////////////
        for(i = 0;i < ROOMWIDTH;i++)
                for(j = 0;j < rowsize;j++)
                        h[currentrank][i][j]=g[currentrank][i][j];

        if(iteration%100==0)
                MPI_Send(&g[currentrank], ROOMWIDTH*rowsize, MPI_DOUBLE, 0,
WORKTAG, MPI_COMM_WORLD);

        if(currentrank%2==0) {  /* even numbered node */
                /////////////////////////////////////////////
                // Send "ghost points" off.
                /////////////////////////////////////////////
                if(currentrank<numprocs-1) {
                        for(i=0;i<ROOMWIDTH;i++)
                                gbelow[i]=g[currentrank][i][rowsize-1];
                        MPI_Send(&gbelow, ROOMWIDTH, MPI_DOUBLE, currentrank + 1,
WORKTAG, MPI_COMM_WORLD);
                        MPI_Recv(&habove, ROOMWIDTH, MPI_DOUBLE, currentrank + 1,
WORKTAG, MPI_COMM_WORLD,&status);
                }
                ////[]/////////////////////////////////////////
                // Send ghost points off
                /////////////////////////////////////////////
                if(currentrank>1) {
                        for(i=0;i<ROOMWIDTH;i++)
                                gabove[i]=g[currentrank][i][0];
                        MPI_Send(&gabove, ROOMWIDTH, MPI_DOUBLE, currentrank-1,
WORKTAG, MPI_COMM_WORLD);
                        MPI_Recv(&hbelow, ROOMWIDTH, MPI_DOUBLE, currentrank-1,
WORKTAG, MPI_COMM_WORLD, &status);

                }
        }
        else {  /* odd numbered node */

                if(currentrank>1) {

                        MPI_Recv(&hbelow, ROOMWIDTH, MPI_DOUBLE, currentrank-1,
WORKTAG, MPI_COMM_WORLD, &status);
                        for(i=0;i<ROOMWIDTH;i++)
                                gbelow[i]=g[currentrank][i][0];


                        MPI_Send(&gbelow, ROOMWIDTH, MPI_DOUBLE, currentrank-1,
WORKTAG, MPI_COMM_WORLD);
                }

                if(currentrank<numprocs-1) {
                        for(i=0;i<ROOMWIDTH;i++) {
                                gabove[i]=g[currentrank][i][rowsize-1];
                        }
                        MPI_Recv(&habove,ROOMWIDTH,MPI_DOUBLE, currentrank+1,
WORKTAG, MPI_COMM_WORLD,&status);
                        MPI_Send(&gabove,ROOMWIDTH, MPI_DOUBLE,
currentrank+1,WORKTAG, MPI_COMM_WORLD);
                }
        }
```

```
                // check if each pixel has converged
                for(i = 0;i < ROOMWIDTH;i++) {
                        for(j = 0;j < rowsize;j++) {
                terminate = TRUE;
                                if(converged[i][j]==FALSE) {
                                        terminate = FALSE;
                                }
                        }
                }
                ////////////////////////////////////////////////////////
                // If each pixel has converged, then we send a message
                // to the host and ask if its time to die.
                ///////////////////////////////////////////////////////
                if(terminate==TRUE) {
                        ///////////////////////////////////////////////////////
                        // This node is ready to go to Node heaven.
                        ///////////////////////////////////////////////////////
                        MPI_Send(&terminate, 1, MPI_INT, 0, QUITTAG, MPI_COMM_WORLD);
                        ///////////////////////////////////////////////////////
                        // Tell me, are the rest of my siblings ready to die?
                        ///////////////////////////////////////////////////////
                        MPI_Recv(&terminate,1,MPI_INT,0,QUITTAG,MPI_COMM_WORLD,&status);
                }
                iteration++;
                if(iteration > limit)
                        terminate = TRUE;
        }  /* end of while (!terminate) */
        terminate = FALSE;
        gterminate = FALSE;
        /* SEND TO  MASTER */
        MPI_Send(&g[currentrank], ROOMWIDTH*rowsize, MPI_DOUBLE, 0, QUITTAG,
MPI_COMM_WORLD);

        while(terminate==FALSE) {
                MPI_Recv(&gterminate, 1, MPI_INT, 0, WORKTAG, MPI_COMM_WORLD, &status);
                if(gterminate==TRUE) {
                        terminate = TRUE;
                }
        }

}


//////////////////////////////////////////////////////////////////////
//      function:       PartitionSize
//      parameters:     numprocs, number of total processors
//      Purpose;        To tell everyone what they are supposedly
//                      working on.
//                      Partition by rows.
//////////////////////////////////////////////////////////////////////
void PartitionData(int numprocs) {

        int i,j;
        int terminate=0;
        int count = numprocs-1;
        int quitsig = 1;
        int globalconvergence = 0;
        int constint = 0;

        MPI_Status status;
        unsigned long rowsize = ROOMHEIGHT/(numprocs-1);
        int converged[60]={0};
        double h[ROOMWIDTH][rowsize];
        unsigned long lastproc = 0;

        while (terminate==FALSE) {

                MPI_Recv(&h, rowsize*ROOMWIDTH, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG,
MPI_COMM_WORLD, &status);
```

```
                //we have convergence, ladies and gentlemeny
                converged[status.MPI_SOURCE] = 1;
                count--;
        if(status.MPI_TAG==QUITTAG) {
        }
        else {

                for(i = 0;i < ROOMWIDTH;i++) {
                        for(j = 0;j < rowsize;j++) {

                                if(h[i][j]<=10.0)
                                        XSetForeground(dpy, gc, BLACK);
                                else if(h[i][j]<=20.0 && h[i][j]>10.0)
                                        XSetForeground(dpy, gc, BLUE);
                                else if(h[i][j]<=30.0 && h[i][j] > 20.0)
                                        XSetForeground(dpy, gc, PURPLE);
                                else if(h[i][j]<=40.0 && h[i][j]>30.0)
                                        XSetForeground(dpy, gc, GREEN);
                                else if(h[i][j]<=50.0 && h[i][j]>40.0)
                                        XSetForeground(dpy, gc, LIGHTGREEN);
                                else if(h[i][j]<=60.0 && h[i][j]>50.0)
                                        XSetForeground(dpy, gc, YELLOW);
                                else if(h[i][j]<=80.0 && h[i][j]>60.0)
                                        XSetForeground(dpy, gc, PINK);
                                else
                                        XSetForeground(dpy, gc, RED);

                                XDrawPoint(dpy,w,gc, i,j + ((status.MPI_SOURCE-
1)*rowsize));
                        }
                }
        }

        if(count==0) {
                globalconvergence=1;
                for(i=1;i<numprocs-1;i++) {
                        if(converged[i]==0) {
                                globalconvergence = 0;
                        }
                        converged[i]=0;
                }
                if(globalconvergence==1) {
                        // We are totally done, let's exit.
                        // now we neeed to do something.
                        for(i=1;i<numprocs-1;i++) {
                                MPI_Send(&globalconvergence,1,MPI_INT,i,QUITTAG,
MPI_COMM_WORLD);
                        }
                }
                else {
                        // Not done, go back and finish please.
                        for(i=1;i<numprocs-1;i++) {
                                MPI_Send(&globalconvergence,1,MPI_INT,i,QUITTAG,
MPI_COMM_WORLD);
                        }
                }
                // reset the counter
                count=numprocs-1;
        }
        ///////////////////////////////////////////////////////
        // NOTE: GLOBAL CONVERGENCE CHECK HERE!!!
        ///////////////////////////////////////////////////////
        if(globalconvergence==1) {
                for(i=1;i<numprocs;i++) {
                        MPI_Send(&quitsig,1,MPI_INT, i, WORKTAG, MPI_COMM_WORLD);
                        quitsig=1;
                }
                global_term=TRUE;
                terminate = TRUE;
                return;
```

```
            }
}
                        }
```

# *References*

*As always, the textbook was used for equations and guidance of how to parallelize the nbody and heat distribution algorithms.*

### *Heat Distribution*

"Two Dimensional Heat Distribution Problem (in a slab of homogeneous material)",
Hewlett Packard, http://www.hp.com/techservers/tutorials3/hpf0014.html#near_jacobi