CS 4444 Parallel Computing Spring 2017

Homework 3: Multicomputer Heated Plate

Eamon Collins

When run, my code successfully performs a simulation of a heated plate to the specifications laid out in submitjob.slurm. The maximum speedup attained was at 200 processors, 1 iteration per cell, and 1 boundary layer, performing 17.54 times better than the sequential version.

Approach:

I decomposed the problem 1-dimensionally along the y axis, chunking the problem into groups of rows. Each processor starts off by allocating and initializing only their portion of the overall grid for both timesteps. Their portion consists of the rows that they are responsible for updating as well as the left and right boundary conditions associated with those rows, and also a layer of ghost cells on top and bottom that will be filled with data from adjacent processors to allow each processor to know what the temperature of the cells is beyond it's own chunk, which it will need to update the cells at the boundary of its chunk. The number of rows in each layer of ghost cells is equal to the boundary_thickness parameter, and will allow each processor to complete as many iterations without communication as there are rows in the layer. The chunks on the top and bottom of the plate also have two layers of ghost cells each, but the layer that is not adjacent to another chunk is fully set to be equal to the corresponding boundary condition temperature each time the processors communicate.

Assigning the processors to the chunks is done top to bottom, with processor 0 taking the top chunk, processor 1 the one below that and so on. This allows each processor to easily find the processors responsible for the chunks above and below it by simply subtracting or adding one respectively, making communication easy.

The communication that takes place between the processors is sending the top and bottom $boundary_thickness rows of each processor's responsibility, and receiving the rows from the adjacent processors into the ghost cell layers. I do this asynchronously, so that all communications on each processor can start before the first one finishes. With asynchronous communication, it is necessary to use an MPI_Wait to make sure that communication has been completed and the data can be modified without causing indeterministic behavior. However, since here each processor is sending and receiving four disjoint areas of its array on each communication, I can place that Wait after all the communications have been started, but before the loops that will need the correct values start.

After each communication, the cells are updated $boundary_thickness times without communication. On the first update in this loop, the algorithm updates all the rows in its matrix except the first and last rows. This will include rows of ghost cells if $boundary_thickness is greater than 1. On each subsequent iteration of this, the rows you update are constricted by one on the top and bottom, until only the cells which are the processors responsibility are updated.

The hotspot is reset every time the plate is updated, and care is taken to update it not only in the processor which is responsible for it but also in the adjacent processor that might have it in one of its ghost cells.

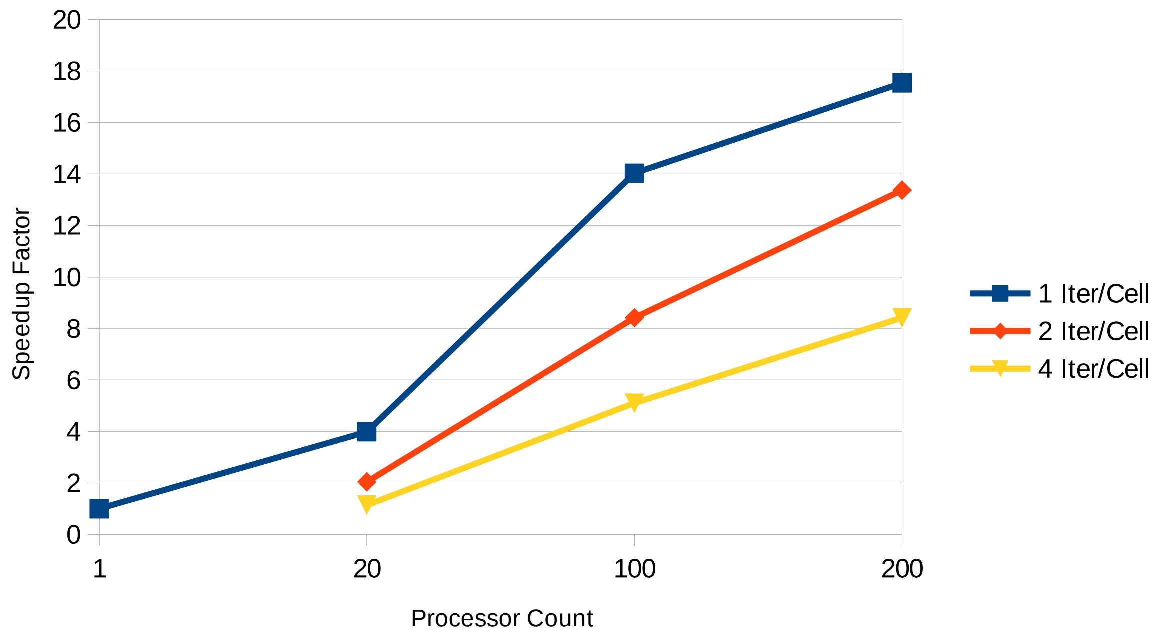
When it comes time to take a snapshot, processor 0 allocates a matrix the size of the whole plate with no extra boundary or ghost cells. All other processors send in the cells they were responsible for updating, and processor 0 asynchronously receives them all into the matrix it created for the snapshot. It then passes that to create_snapshot.

Results:
All results reported are for a plate size of 10000x10000 over 10000 iterations.
All binaries used, including the sequential version, have been compiled with -O3
Speedup listed is as compared to the sequential version given to us.

Processors	**Iters per Cell**	**Boundary Thickness**	**Execution Time (sec)**	**Speedup**
1	1	N/a	842	1
20	1	1	211	3.99
20	1	2	212	3.97
20	1	4	213	3.95
20	2	1	413	2.04
20	2	2	411	2.05
20	2	4	412	2.04
20	4	1	739	1.14
20	4	2	741	1.14
20	4	4	741	1.14
100	1	1	60	14.03
100	1	2	61	13.8
100	1	4	58	14.5
100	2	1	100	8.42
100	2	2	102	8.25
100	2	4	101	8.34
100	4	1	165	5.1
100	4	2	166	5.07
100	4	4	170	4.95
200	1	1	48	17.54
200	1	2	111	7.59 (outlier)
200	1	4	52	16.19
200	2	1	63	13.37
200	2	2	72	11.69
200	2	4	73	11.53
200	4	1	100	8.42

200	4	2	108	7.8
200	4	4	109	7.72

The below graph gives a bit better illustration of the effect the processor count and iters_per_cell had on the speedup. Graphed values are for boundary thickness values of 1, and I did not include those on the graph because varying the boundary thickness did not produce statistically separable differences in speedup. I discuss why that might be below.



Effect of Processor Count on Speedup

Analysis:

From the results, we see that the number of processors greatly improves speedup for each level of processor counts tested. Doubling the amount of iters_per_cell decreased the speedup each time, although not quite by half in all cases. Increasing the thickness of the ghost cell layers had very little noticeable effect on speedup, and if anything increased it slightly.

First to clear up the difference between iters_per_cell and boundary_thickness: the iters/cell count is a way to effectively increase the granularity of the problem, that is the ratio of processing done to the total amount of communication that takes place. It does this by increasing the amount of work without affecting the amount of communication at all. The boundary_thickness, however, allows you to do more processing in between communication, but the amount of data that must be sent at every communication increases by the same factor. Theoretically this would cut down on the overhead induced by sending each message, but not the cost associated with the amount of data that needs to be sent.

Looking at the graph, the improvement between 20 to 100 processors is much greater than that between 100 and 200. Even controlling for the fact that you are multiplying the processors by 5 in the first step and 2 in the second, this disparity persists. This means the inflection point, where the message cost is balanced by the amount of processing with neither providing a bottleneck, occurs somewhere between processor 20 and 100. Before the inflection point, the bottleneck is the processing cost, after the inflection point the bottleneck is the messaging cost incurred by many processors who all need to send and receive messages. This conclusion is bolstered by the fact that as we increase the granularity (done by increasing the iters/cell,) the speedup cost of doubling the actual work we do goes down. That is, at 20 processors, doubling the work without touching communication results in doubling the execution time. That means the bottleneck at that point is in the processing. However, at 200 processors, doubling the amount of work done total only increases the execution time by a factor of 1.3. This is because the bottleneck is initially the communication, and we have to increase the work done by a lot to put the bottleneck back at the processing.

After the inflection point, adding each additional processor still improves speedup, but it improves it by less each processor. This will continue until some point after 200 processors, where the message cost becomes so high that adding additional processors will actually start to decrease the speedup received.

As to the boundary_thickness, I believe it's seeming non-inclusion in the results is both a testament to the low overhead cost of each message and also the extra work it incurs. Increasing it will indeed reduce the total number of messages sent and therefore the total latency cost, but it also requires each processor to do redundant work. Each processor calculating and updating its ghost cells takes time, but that work is effectively discarded after it is used to update the cells the processor is actually responsible for. In fact, this could explain the slight difference in speedup at higher processors. At 20 processors, each processor is responsible for updating 500 rows, and the ghost cells are a drop in the bucket. However, at 200 processors, each processor is only responsible for 50 rows. A boundary_thickness of 4 adds an average of 3 redundant rows to be calculated and updated to each. (Average calculation: ((3+2+1)/4)*2.) That's 6% of the rows that processor is responsible for, and could explain why at higher processor counts there is a slight upward tick in execution time for higher boundary_thickness, while at 20 it seems to be exactly the same.

Conclusions:

To find the "sweet spot" of the program, achieving the highest parallelization efficiency, or contribution to speedup per processor, I would start with 100 processors and incrementally increase the granularity by increasing the problem size until the speedup started to decrease. We know there is room to increase the granularity at this number of processors because doubling the iters/cell did not double execution time. After that, if I wished to find the maximum useful number of processors, I would simply increase the number of processors until the speedup actually started to decrease.

Assuming we want to hold the problem size constant, it appears that the most efficient granularity will be found somewhere in between 100 and 200 processors, closer to 100, while the best possible speedup will be somewhere past 200 processors.

RELEVANT FILES

halo.c – the code for my mpi version of heated_plate
Makefile – my makefile to make the halo binary
submitjob.slurm – my submission script for specifying parameters and submitting the job.
Snapshot.10000.jpeg – the snapshot produced by my code's run at the 10000th iteration
output – directory with all the output files from my benchmarking runs.

HOW TO RUN MY CODE:
log onto rivanna with submitjob.slurm, halo.c, and the Makefile
module load mvapich2/gcc
make
set parameters you want in submitjob.slurm. Currently set for a 200-1-1 of procs-iters/cell-boundary thickness
sbatch submitjob.slurm
when done check the output file that is created.

PLEDGE:
On my honor as a student, I have neither given nor received unauthorized aid on this assignment.
-Eamon Collins

SUBMITJOB.SLURM

```
#!/bin/bash
#SBATCH --nodes=10
#SBATCH --ntasks=200
#SBATCH --ntasks-per-node=20
#SBATCH --cpus-per-task=1
#SBATCH --mem-per-cpu=1096
#SBATCH --time=02:00:00
#SBATCH --part=eqa-cs6414
#SBATCH --account=parallelcomputing
#SBATCH --output=output200-1-4

module load mvapich2/gcc
mpiexec ./halo 200 1 10000 10000 4
```

HALO.C

```
// This program simulates the flow of heat through a two-dimensional plate.
// The number of grid cells used to model the plate as well as the number of
//  iterations to simulate can be specified on the command-line as follows:
//  ./heated_plate_sequential <columns> <rows> <iterations>
// For example, to execute with a 500 x 500 grid for 250 iterations, use:
//  ./heated_plate_sequential 500 500 250


#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <time.h>
#include "mpi.h"


// Define the immutable boundary conditions and the inital cell value
#define TOP_BOUNDARY_VALUE 0.0
#define BOTTOM_BOUNDARY_VALUE 100.0
#define LEFT_BOUNDARY_VALUE 0.0
#define RIGHT_BOUNDARY_VALUE 100.0
#define INITIAL_CELL_VALUE 50.0
#define hotSpotRow 4500
#define hotSptCol 6500
#define hotSpotTemp 1000
#define num_cols 10000
#define num_rows 10000


// Function prototypes
```

```
void print_cells(float **cells, int n_x, int n_y);
void initialize_cells(float **cells, int n_x, int n_y, int n_b);
void create_snapshot(float **cells, int n_x, int n_y, int id);
float **allocate_cells(int n_x, int n_y);
void die(const char *error);


int main(int argc, char **argv) {
	// Record the start time of the program
	time_t start_time = time(NULL);

	// Extract the input parameters from the command line arguments
	// Number of columns in the grid (default = 1,000)
	// int num_cols = (argc > 1) ? atoi(argv[1]) : 1000;
	// // Number of rows in the grid (default = 1,000)
	// int num_rows = (argc > 2) ? atoi(argv[2]) : 1000;
	// // Number of iterations to simulate (default = 100)
	// int iterations = (argc > 3) ? atoi(argv[3]) : 100;
	//number of chunks in x dimension
	int y_dim = (argc > 1) ? atoi(argv[1]) : 1;
	//iters_per_cell
	int iters_per_cell = (argc > 2) ? atoi(argv[2]) : 1;
	int iters_per_snap = (argc > 3) ? atoi(argv[3]) : 1000;
	// Number of iterations to simulate (default = 100)
	int iterations = (argc > 4) ? atoi(argv[4]) : 100;
	int boundary_thickness = (argc > 5) ? atoi(argv[5]) : 1;

	int id;
	int p;
	//start up mpi. WHoooo!
	MPI_Init(&argc, &argv);
	MPI_Comm_rank(MPI_COMM_WORLD, &id);
	MPI_Comm_size(MPI_COMM_WORLD, &p);
	if (id == 0){
		// Output the simulation parameters
		printf("Grid: %dx%d, Iterations: %d\n", num_cols, num_rows, iterations);
		printf("procs: %d\niterspercell: %d\nboundarythick:
%d",p,iters_per_cell,boundary_thickness);
		fflush(stdout);
	}
	//barrier in case there's something odd with the program start, eg slurm doesn't
requisition all processors immediately or something
	MPI_Barrier(MPI_COMM_WORLD);
	//Check if the number of processors specified in the run command lines up with the
SLURM specified procs
	if(y_dim == p){

		//want quick access to number of problem rows per processor chunk
		int rows_each = num_rows/y_dim;
		// We allocate two arrays: one for the current time step and one for the next time
step.
		// At the end of each iteration, we switch the arrays in order to avoid copying.
		// The arrays are allocated with an extra surrounding layer which contains
		//  the immutable boundary conditions (this simplifies the logic in the inner loop).
		//
		//each matrix is allocated with 1 extra on the left and right for the boundary
conditions, as well as
		//a layer on the top and bottom of boundary_thickness cells for use in iterating
		float **cells[2];
		cells[0] = allocate_cells(num_cols + 2, rows_each + boundary_thickness*2);
		cells[1] = allocate_cells(num_cols + 2, rows_each + boundary_thickness*2);
		int cur_cells_index = 0, next_cells_index = 1;

		// Initialize the interior (non-boundary) cells to their initial value.
		// Note that we only need to initialize the array for the current time
		//  step, since we will write to the array for the next time step
		//  during the first iteration.
		initialize_cells(cells[0], num_cols, rows_each, boundary_thickness);
		// Set the immutable boundary conditions in both copies of the array
		int x, y, i, j, b, ic;
		//the top and bottom procs also have 2 boundary_thickness buffers, they are just
all allocated to the boundary values.
		if (id == 0){
			for(j = 0; j < boundary_thickness; j++){
				for (x = 0; x <= num_cols+1; x++) cells[0][j][x] = cells[1][j][x] =
TOP_BOUNDARY_VALUE;
			}
		}
		if (id == p-1){
			for(j = 0; j < boundary_thickness; j++){
				for (x = 0; x <= num_cols+1; x++) cells[0][rows_each + 1 + j][x] =
cells[1][rows_each + 1 + j][x] = BOTTOM_BOUNDARY_VALUE;
			}
		}
		for (y = 0; y < rows_each + boundary_thickness*2; y++) cells[0][y][0] = cells[1][y]
[0] = LEFT_BOUNDARY_VALUE;
		for (y = 0; y < rows_each + boundary_thickness*2; y++) cells[0][y][num_cols + 1]
= cells[1][y][num_cols + 1] = RIGHT_BOUNDARY_VALUE;


		//set the hotspot before starting
		if(num_cols >= hotSptCol && rows_each*id <= hotSpotRow &&
rows_each*(id+1) > hotSpotRow){
			cells[cur_cells_index][hotSpotRow % rows_each + boundary_thickness]
[hotSptCol+1]= (float)hotSpotTemp;
		}

		// Simulate the heat flow for the specified number of iterations
		//divide by boundary thickness so you have a place to communicate and know
how many iterations it has been since refresing ghost cells
		for (i = 0; i < iterations / boundary_thickness; i++) {
			//send and receive ghost cells from "neighboring" processors
			//I send and receive asynchronously, using the cells matrices as the
buffers. I don't need to wait in the middle as the cells that form the ghost cells for the
			//adjacent processor are part of the current processors actual cells, while
its ghost cells will be above or below that so there is no chance of writing data as it is
			//being loaded into a buffer to be sent.
			//Additionally, since all my comms are async, there is no real need to be
aware of the possibility of zippering, but I flipped my send/recvs on even/odd processors just
in case
			//and because it might be slightly faster to have the commands that
communicate with each other execute at the same time.
			MPI_Request top_send, bottom_send, top_recv, bottom_recv;
			if (id == 0){
				MPI_Isend(&cells[cur_cells_index][rows_each][0],
boundary_thickness*(num_cols + 2), MPI_FLOAT, id + 1, id, MPI_COMM_WORLD,
&bottom_send);
				MPI_Irecv(&cells[cur_cells_index][boundary_thickness +
rows_each][0], boundary_thickness*(num_cols + 2), MPI_FLOAT, id + 1, id + 1,
MPI_COMM_WORLD, &bottom_recv);
				for (y = 0; y < boundary_thickness; y++)
					for(x = 0; x < num_cols + 2; x++)
						cells[cur_cells_index][y][x] =
TOP_BOUNDARY_VALUE;
				//does not need to send or receive the boundary condition ghost
cells, so just set the request as null to make the code easier
				top_send = MPI_REQUEST_NULL;
				top_recv = MPI_REQUEST_NULL;
			}else if (id == p-1){
				MPI_Irecv(&cells[cur_cells_index][0][0],
boundary_thickness*(num_cols + 2), MPI_FLOAT, id -1, id - 1, MPI_COMM_WORLD,
&top_recv);
				MPI_Isend(&cells[cur_cells_index][boundary_thickness][0],
boundary_thickness*(num_cols + 2), MPI_FLOAT, id -1, id, MPI_COMM_WORLD,
&top_send);
				for (y = 0; y < boundary_thickness; y++)
					for(x = 0; x < num_cols +2; x++)
						cells[cur_cells_index][y][x] =
BOTTOM_BOUNDARY_VALUE;
				bottom_send = MPI_REQUEST_NULL;
				bottom_recv = MPI_REQUEST_NULL;
			}else if (id % 2 == 0){
				MPI_Isend(&cells[cur_cells_index][boundary_thickness][0],
boundary_thickness*(num_cols + 2), MPI_FLOAT, id -1, id, MPI_COMM_WORLD,
&top_send);
```

```
                MPI_Isend(&cells[cur_cells_index][rows_each][0],
boundary_thickness*(num_cols + 2), MPI_FLOAT, id + 1, id, MPI_COMM_WORLD,
&bottom_send);
                MPI_Irecv(&cells[cur_cells_index][0][0],
boundary_thickness*(num_cols + 2), MPI_FLOAT, id - 1, id - 1, MPI_COMM_WORLD,
&top_recv);
                MPI_Irecv(&cells[cur_cells_index][boundary_thickness +
rows_each][0], boundary_thickness*(num_cols + 2), MPI_FLOAT, id + 1, id + 1,
MPI_COMM_WORLD, &bottom_recv);
            }else if ( id % 2 == 1){
                MPI_Irecv(&cells[cur_cells_index][boundary_thickness +
rows_each][0], boundary_thickness*(num_cols + 2), MPI_FLOAT, id + 1, id + 1,
MPI_COMM_WORLD, &bottom_recv);
                MPI_Irecv(&cells[cur_cells_index][0][0],
boundary_thickness*(num_cols + 2), MPI_FLOAT, id - 1, id - 1, MPI_COMM_WORLD,
&top_recv);
                MPI_Isend(&cells[cur_cells_index][boundary_thickness][0],
boundary_thickness*(num_cols + 2), MPI_FLOAT, id - 1, id, MPI_COMM_WORLD,
&top_send);
                MPI_Isend(&cells[cur_cells_index][rows_each][0],
boundary_thickness*(num_cols + 2), MPI_FLOAT, id + 1, id, MPI_COMM_WORLD,
&bottom_send);
            }
            //wait to make sure all data is ready for the next iteration and securely in
the cells matrix
            MPI_Wait(&top_send, MPI_STATUS_IGNORE);
            MPI_Wait(&bottom_send, MPI_STATUS_IGNORE);
            MPI_Wait(&top_recv, MPI_STATUS_IGNORE);
            MPI_Wait(&bottom_recv, MPI_STATUS_IGNORE);

            //iterate using the ghost cells to update the problem cells. You can update
once for each ghost cell.
            for (b = 0; b < boundary_thickness; b++){
                //each time you update, a ghost cell on the fringe becomes
invalidated and should no longer be updated as it will be affected by
                //the boundary of the ghost cells for which we do not have data.
Therefore constrict the loop boundaries one ghost cell down and
                //one ghost cell up each time you update until you get a fresh batch
of ghost cells, then start over again
                for (y = 1 + b; y <= rows_each + boundary_thickness*2 - b - 2; y++)
{
                    for (x = 1; x <= num_cols; x++) {
                        //iters per cell loop to simulate a larger problem being
doen with the same amount of communication
                        //ie larger granularity
                        //I considered factoring the array accesses out of the
iters_per_cell loop body, but figured
                        //that the array accesses each time are part of the
artificial work we are inserting.
```

```
						for(ic = 0; ic < iters_per_cell; ic++){
							// The new value of this cell is the average of
the old values of this cell's four neighbors
							cells[next_cells_index][y][x] =
(cells[cur_cells_index][y][x - 1]  +

cells[cur_cells_index][y][x + 1]  +

cells[cur_cells_index][y - 1][x]  +

cells[cur_cells_index][y + 1][x]) * 0.25;
						}
					}
				}
				// Swap the two arrays
				cur_cells_index = next_cells_index;
				next_cells_index = !cur_cells_index;
				//if the hotspot is within the current problem size, set it to its temp
				//need to set it's temp both on the processor that has it in the cells
it is responsible for as well
				//as in the ghost cells of the neighboring processor.
				if(num_cols >= hotSptCol && rows_each*id <= hotSpotRow &&
rows_each*(id+1) > hotSpotRow){
					cells[cur_cells_index][hotSpotRow % rows_each +
boundary_thickness][hotSptCol+1]= (float)hotSpotTemp;
				}else if(num_cols >= hotSptCol && rows_each*(id+1) <=
hotSpotRow && rows_each*(id+2) > hotSpotRow && (hotSpotRow % rows_each) <
boundary_thickness){
					cells[cur_cells_index][hotSpotRow % rows_each +
boundary_thickness + rows_each][hotSptCol+1] = (float)hotSpotTemp;
				}else if(num_cols >= hotSptCol && rows_each*(id-1) <=
hotSpotRow && rows_each*(id) > hotSpotRow && rows_each - (hotSpotRow % rows_each)
<= boundary_thickness){
					cells[cur_cells_index][rows_each-(hotSpotRow %
rows_each) -1][hotSptCol+1] = (float)hotSpotTemp;
				}
			}

			// Print the current progress every time you communicate
			// I commented this out to benchmark, mostly because it was annoying
when each output extended past the limit of the terminal
			// but it might also have minor speed effects.
			/**
			if(id==0){
				printf("Iteration: %d / %d\n", i*boundary_thickness + 1, iterations);
				fflush(stdout);
			}**/
```

```
            //every iterations_per_snapshot you have to collect all the separate
chunks into one matrix and create a snapshot from it
            if (((i+1)*boundary_thickness) % iters_per_snap == 0 && i > 0){
                int final_cells = (iterations % 2 == 0) ? 0 : 1;
                if (id != 0){
                    //send only the cells this processor is actually responsible
for, no ghost cells or boundary conditions
                    MPI_Send(&cells[final_cells][boundary_thickness][0],
(num_cols+2)*(rows_each), MPI_FLOAT, 0, id, MPI_COMM_WORLD);
                }else{
                    float **all_cells;
                    all_cells = allocate_cells(num_cols+2, num_rows+2);

                    // copy in one top boundary layer plus all the problem rows
in processor 0
                    memcpy(all_cells[0], cells[final_cells][boundary_thickness-
1], (num_cols+2)*(rows_each+1)*sizeof(float));
                    //init bottom boundary layer
                    for (x = 0; x <= num_cols+1; x++) all_cells[rows_each + 1][x]
= BOTTOM_BOUNDARY_VALUE;
                    int l;
                    //need a request object for each processor minus proc 0
                    MPI_Request reqs[y_dim -1];
                    //loop over all procs except proc 0 as we already have
access to its cells
                    for (l=1; l < y_dim; l++){
                        MPI_Irecv(all_cells[l*rows_each+1],
(num_cols+2)*(rows_each), MPI_FLOAT, l, l,MPI_COMM_WORLD, &reqs[l-1]);
                    }
                    // wait for all the cells to be successfully received
                    MPI_Waitall(y_dim-1, reqs, MPI_STATUSES_IGNORE);

                    // Output a snapshot of the final state of the plate
                    create_snapshot(all_cells, num_cols, num_rows,
(i+1)*boundary_thickness);
                    free(all_cells);
                }
            }
            //need a barrier every iteration so the processors are not on different
iterations when they communicate
            // (but not every boundary_thickness dependent iteration as they don't
need to communicate then)
            MPI_Barrier(MPI_COMM_WORLD);
        }

        free(cells[0]);
        free(cells[1]);

        // Compute and output the execution time
```

```
		time_t end_time = time(NULL);
		printf("\nExecution time: %d seconds\n", (int) difftime(end_time, start_time));
		fflush(stdout);
	}else{
		if(id == 0){
			printf("number of processors in slurm file does not match the
decomposition parameter on the command line");
			fflush(stdout);
		}
	}
	MPI_Finalize();
	return 0;
}


// Allocates and returns a pointer to a 2D array of floats
float **allocate_cells(int cols, int rows) {
	float **array = (float **) malloc(rows * sizeof(float *));
	if (array == NULL) die("Error allocating array!\n");

	array[0] = (float *) malloc(rows * cols * sizeof(float));
	if (array[0] == NULL) die("Error allocating array!\n");

	int i;
	for (i = 1; i < rows; i++) {
		array[i] = array[0] + (i * cols);
	}

	return array;
}


// Sets all of the specified cells to their initial value.
// Assumes the existence of a one-cell thick boundary layer.
// Sets all values in array to the initial cell values, boundary conditions will be reset later
void initialize_cells(float **cells, int cols, int rows, int boundary_thickness) {
	int x, y;
	for (y = 0; y < rows + boundary_thickness*2; y++) {
		for (x = 1; x <= cols; x++) {
			cells[y][x] = INITIAL_CELL_VALUE;
		}
	}
}


// Creates a snapshot of the current state of the cells in PPM format.
// The plate is scaled down so the image is at most 1,000 x 1,000 pixels.
// This function assumes the existence of a boundary layer, which is not
//  included in the snapshot (i.e., it assumes that valid array indices
//  are [1..num_rows][1..num_cols]).
void create_snapshot(float **cells, int cols, int rows, int id) {
        int scale_x, scale_y;
        scale_x = scale_y = 1;

        // Figure out if we need to scale down the snapshot (to 1,000 x 1,000)
        //  and, if so, how much to scale down
        if (cols > 1000) {
                if ((cols % 1000) == 0) scale_x = cols / 1000;
                else {
                        die("Cannot create snapshot for x-dimensions >1,000 that are not
multiples of 1,000!\n");
                        return;
                }
        }
        if (rows > 1000) {
                if ((rows % 1000) == 0) scale_y = rows / 1000;
                else {
                        printf("Cannot create snapshot for y-dimensions >1,000 that are not
multiples of 1,000!\n");
                        return;
                }
        }

        // Open/create the file
        char text[255];
        sprintf(text, "snapshot.%d.ppm", id);
        FILE *out = fopen(text, "w");
        // Make sure the file was created
        if (out == NULL) {
                printf("Error creating snapshot file!\n");
                return;
        }

        // Write header information to file
        // P3 = RGB values in decimal (P6 = RGB values in binary)
        fprintf(out, "P3 %d %d 100\n", cols / scale_x, rows / scale_y);

        // Precompute the value needed to scale down the cells
        float inverse_cells_per_pixel = 1.0 / ((float) scale_x * scale_y);

        // Write the values of the cells to the file
        int x, y, i, j;
        for (y = 1; y <= rows; y += scale_y) {
                for (x = 1; x <= cols; x += scale_x) {
                        float sum = 0.0;
                        for (j = y; j < y + scale_y; j++) {
                                for (i = x; i < x + scale_x; i++) {
                                        sum += cells[j][i];
                                }
                        }
                        // Write out the average value of the cells we just visited
                        int average = (int) (sum * inverse_cells_per_pixel);
                        //otherwise hotspot makes it weird
                        if (average > 100 ) average = 100;
                        fprintf(out, "%d 0 %d\t", average, 100 - average);
                }
                fwrite("\n", sizeof(char), 1, out);
        }

        // Close the file
        fclose(out);
}


// Prints the specified error message and then exits
void die(const char *error) {
        printf("%s", error);
        exit(1);
}
```