CS 4444 – Report for Homework 3

Jerry Sun (ys7va)

1 Problem Description

The goal of this assignment is to write a parallel C program using openmpi to optimize the runtime for a lightly twisted classic heated plate problem, and to discover the performance regarding several concerns around parallel program like overhead, efficiency, etc.

The halo problem I am dealing with here has the setting as below:

- The interior cells should all initialized to the same temperature (50 degrees)
- The border cells fixed at a specific temperature (0 degrees along the top and left sides and 100 degrees along the bottom and right sides)
- There is a single internal condition (col=6500 & row=4500) in which a single cell is held at 1000 degrees
- In each time step of the simulation, the temperature of each cell is computed by averaging the temperatures of the four neighboring cells in the previous time step

The original sequential program provided has a runtime of 1671 seconds for a 10000 iterations on $10000 * 10000$ plates

2 Result Summary

For 10000 iterations on $10000 * 10000$ plates with the hotspot, my final parallel program works, and gives the best performance with 2 ghost cell layers which takes 27 seconds to finish including the final image rendering. The overall speedup is 67 times faster than the original sequential program.

3 Metadata

3.1 Software

SLURM: version 14.11

GNU bash: version 4.1.2

openmpi: version 1.8.4

gcc: version 4.4.7

3.2 Hardware

10 core Intel(R) Xeon(R) CPU E5-2670 v2 @ 2.50GHz with a cache size of 25600KB

4 Approach

For this assignment, it is required to parallelize the particular halo problem described above. There are several main core designs I use to implement the parallelized version of halo.

4.1 Divide & Ghost Cell

For a given number of nodes and a fixed size plate, I want to divide the whole plate evenly onto each node, so that it can distribut the computation evenly across different nodes. Here, in particular I divide the whole plate in a row-major order, as each node will be responsible for a number of continuous rows with all columns. The reason and the advantage of this approach can be found in Optimization section later.

However, if each node only maintain the cells that it needs to compute, the result will not be accurate, since the outer cells might need the information from neighbor nodes to maintain their accuracy. Therefore, I add the ghost cells around the core block. These ghost cells will receive the value of the corresponding cells in the neighboring nodes, thus providing the center block values to compute with.

Unfortunately, the other problem is that the ghost cells will still loose its accuracy since they are now on the boundary. Therefore, the program need to routinely update the ghost cells, and the thickness of the ghost cells directly determines how often do does it need to do the update(detail see next subsection).

4.2 Message Passing

Message passing is extremely tricky to deal with in parallel program. We want to maintain the accuracy, while achieving a higher efficiency(also avoiding deadlocks) since the overhead of message passing between nodes will deprecate the performance of the whole program especially when there is a large number of nodes, like 200.

The approach here I need to consider includes two parts, first is how often does the program need to send the message, next is how is it going to send values.

First, for a fixed thickness of the ghost cell, say n cells thick, the program can maintain its accuracy in the "core blocks" for n iterations, since for each iteration, one single row of cells, from edge to the center, will lose its accuracy. Therefore it only need to transfer the value of cells every n iterations instead of every single iterations.

Second, the most important aspect regarding message passing is the way to implement the send/receive patterns since it is extremely easy to have deadlock or really inefficient sending pattern. Our design classify all message transfer into four categories as follow:

- tag 0: even node to odd node upward send
- tag 1: odd to even upward send
- tag 2: even to odd downward send
- tag 3: odd to even downward send

Note: upward send just means from lower rank node to higher rank node, and downward send just means the opposite way.

Then for a single odd-rank node, it will perform the following 4 steps:

- receive ghost cells from lower even node
- send ghost cells to higher even node
- receive ghost cells from higher even node
- send ghost cells to lower even node

For a even-rank node then it just perform the other way around.

Note that during implementation, it also needs to separate the first and last node, since they only need to do one send and one receive. For node 0 it is simple, since it only need to handle tag 0 and tag 3. However, for the last node, there are actually two cases. If the number of nodes is even, then the last node has odd rank. Therefore it needs to handle tag 0 and tag 3. However, if the number of nodes is odd, then it needs to handle the other two tags.

After all those steps, our message transfer can be grouped into 4 tags and inside each tag, all message passing can happen simultaneously, thus providing better performance.

5 Performance

In this section, I will give a brief analysis for the data I retrieved from the testing above. For time measurement, I use the *time()* function in 'time.h' library. For timing, it counts the total runtime from the start of the program, to the point right after *MPI_Finalize()*.

Since the clock time of the program running on Rivanna varies, I retrieve the optimal runtime for each condition. Also, the process to create snapshot is related to file write which takes extra time to finish, so for all experiments the program only produces one final snapshot after 10000 iterations.

5.1 Runtime data collection

Followed are all the timing recorded during testing on Rivanna based on different number of tasks, number of iterations per cell and different number of ghost cells.

Table 1: 200 tasks

iter per cell	1 ghost layer	2 ghost layer	3 ghost layer
1	30	27	28
2	50	52	56
4	79	82	88

Table 2: 100 tasks

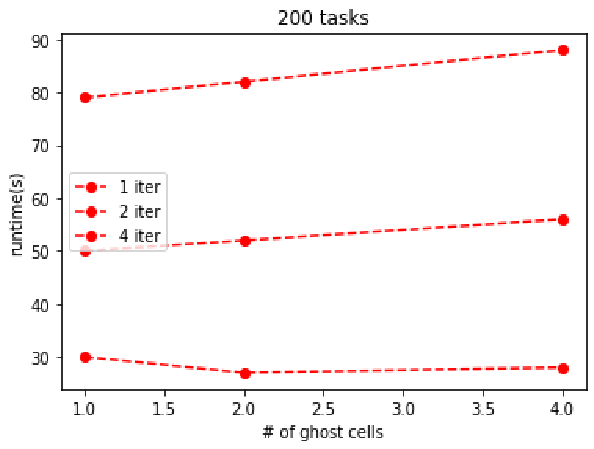
iter per cell	1 ghost layer	2 ghost layer	4 ghost layer
1	56	57	60
2	97	99	103
4	157	159	165

Table 3: 20 tasks

iter per cell	1 ghost layer	2 ghost layer	4 ghost layer
1	266	263	266
2	474	473	477
4	773	774	779

5.2 Ghost Cell Analysis

First, I want to explore the runtime differences between different number of ghost cells. The following graph shows the runtime comparison regarding different number of ghost cells with 200 tasks.



From the graph above we can see that there is no significant difference regarding the performance of different number of ghost cells. This is mainly due to the fact that while a larger number of ghost cells can lead to a smaller number of message passing, the total amount of the data that need to be transferred increases, also to maintain the accuracy of the center blocks the program also need to have more calculations for each node within the extra ghost cells. Both of those two aspects might cancel out the advantage of fewer message passing.

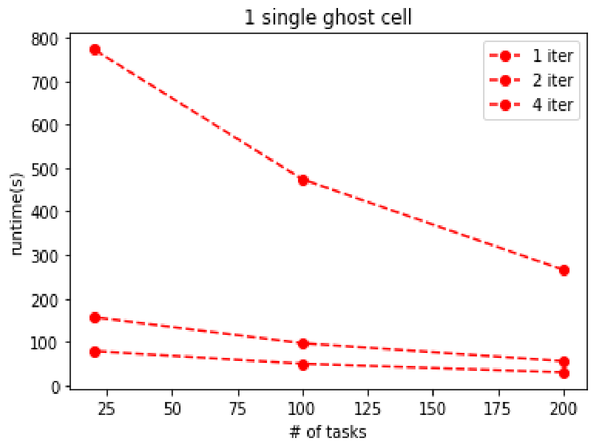
5.3 Granularity Analysis

Second, I want to explore the effect of increasing granularity. If the program only increases the number of computations while maintain any other aspects (simply by executing the "averaging process" multiple times), how will the performance be affected? Here I pick all the results for 1 single ghost cell.

We can find that the growth in runtime is not proportional to the growth in the number of iterations per cell. This is simply because of the overhead due to the message passing. Especially when 200 tasks are deployed, if there is no overhead, the theoretical runtime for 4 iterations per cycle should be $4 * 30 = 120$, but in reality the run time is only 80. This means that a large proportion of the runtime for 1 iteration per cell is caused by overhead, including initiating MPI, message passing, etc.

5.4 Tasks Analysis

Finally, I want to explore the influence of deploying more tasks and the corresponding efficiency score. Here I pick all the results for 1 ghost cell.



From the graph on the left, we can see non-trivial improvement in performance given an increasing number of tasks. While there exists overhead observed at prior sections, I can still state that the performance gain at 200 tasks level is significant.

6 Optimization

6.1 row-major division

Here I prefer to divide the plate into multiple rows with different columns instead of a usual column division technique shown in class/textbook. The main criteria I want to consider for memory alignments are cache locality during computation and the message passing expense. For the cache locality, inside the for loop we typically choose to have the outer loop as rows and the inner loop as columns. Therefore a row-major division can give a decent cache performance by accessing continuous memory. The other advantage row-major division has is that when sending cells' value to other nodes, the program always wants to send continuous memory. Therefore, a row-major data division will be corresponding to the way cell values are stored in the initial/final maps.

6.2 message passing

As discussed earlier in Approach, the message passing design is an extremely important optimization compared to a regular linear send/receive process. For one single processor, the theoretical lower bound for message-passing function it has to call is 4 (two sends and two receives). Our current implementation actually meets this lower bound, as each processor will finish its message passing task without extra wait time because of the design problem.

Also, I have tested asynchronous message passing using ISEND, IRECEIVE in a smaller scale. However, the result shows that there is no significant performance bonus using those two methods compared to the performance using SEND/RECEIVE.

7 conclusion

In this assignment, I have gain solid experience in designing and implementing parallel program using MPI. The parallel program I implemented shows a substantial performance speedup than the original sequential program with a speedup rate over 60 times. Also, the observation made within different circumstances indicates the influence of overhead created by message passing is significant especially in large number of tasks. At the same time, however, simply reducing rate of the message passing by increasing number of ghost cell layers doesn't have a solid improvement for the overall performance because of other disadvantages using this technique.

8 Pledge

On my honor as a student, I have neither received nor given help on this assignment.

Signature: ______________

9 Appendix

9.1 halo.c

```
// This program simulates the flow of heat through a two-dimensional plate in parallel ↩
    using MPI
// sequential version writtern by Andrew Grimshaw
// modified parallel version implemented by Jerry Sun (ys7va)
// 4/12/2017

// input parameters :
// dimension: x, y (size of the plate)
// iter per cell: number of iterations for each computation
// iter per snapshot: number of iterations per snapshot creation
// iterations: total number of iterations to simulates
// ghost: number of ghost-cells
// compile format: mpicc -O3 -o halo halo.c
// execute format: mpirun -np 200 halo 10000 10000 1 10000 10000 2
// this simulate 10000 * 10000 plate with 10000 iterations using 2 ghost-cells within 200 ↩
    tasks

#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <mpi.h>
#include <string.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 hotSpotCol 6500
#define hotSpotTemp 1000



// Function prototypes, comments above only give a brief description,
// detailed explanation and implementation can be found after main method

// directly print the values of the cells given a block, used only for debug purposes
void printMaps(float **cells, int n_x, int n_y,int rank, int type);

// initialize all the cells inside the block with value 50
void initialize_cells(float **cells, int n_x, int n_y, int ghost);

// create a snapshot as ppm file given a cell block
void create_snapshot(float **cells, int n_x, int n_y, int id);

// set the boundary value for a given block for a single node
void set(float ***cells, int n_x, int n_y, int ghost, int rank, int nodes);

// allocate a 2-D array with given number of rows and columns, this 2-D array is
// actually continuous in memory
float **allocate_cells(int n_x, int n_y);

// kill the program
void die(const char *error);
```

```
// set the hotspot into correct node and position
void hotspot(float **cells, int n_x, int n_y, int ghost, int rank, int nodes);

// deal with message passing through different node
void send(float **cells, int ghostSize, int ghost, int numrows, int rank, int nodes);

// main method
int main(int argc, char **argv) {
    // Record the start time of the program
    time_t start_time = time(NULL);
    // variable declaration for various iterator in for loop
    int x, y, i, iter;
    // Extract the input parameters from the command line arguments
    // Number of columns in the grid (default = 1,000)
    int real_cols = (argc > 1) ? atoi(argv[1]) : 1000;

    // Number of rows in the grid (default = 1,000)
    int real_rows = (argc > 2) ? atoi(argv[2]) : 1000;

    // the number of inner loop iterations to compute per cell formed from the chunk ↩
        divisions.
    int iters_per_cell = (argc > 3) ? atoi(argv[3]) : 1000;

    // number of iterations to output the matrix to snapshot.X in which X is the iteration↩
         number.
    int iterations_per_snapshot = (argc > 4) ? atoi(argv[4]) : 1000;

    // Number of iterations to perform (default = 100)
    int iterations = (argc > 5) ? atoi(argv[5]) : 100;

    // boundary_thickness number of ghost cell layers to send at a time and how many ↩
        internal
    // iterations to perform per communication (defaut = 1)
    int ghost = (argc > 6) ? atoi(argv[6]) : 1;

    //initialize MPI
    MPI_Init(&argc, &argv);
    //retrive number of nodes in the whole program
    int nodes;
    MPI_Comm_size(MPI_COMM_WORLD, &nodes);
    // retrive the rank for the current process
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // pointer to the cell block, one for current block, one for next
    // we use them alternatively for computaion
    float **cells[2];

    // pointer to a whole final image, used for final image printout
    float **final_result;

    // num_cols and num_rows represents the number of rows/columns
    // for a single node that it need to handle
    int num_cols = real_cols;
    int num_rows = real_rows/nodes;

    // total_rows/cols represents the number of rows/cols that one node need to
    // store, the difference between total vs num comes from the fact that a node
    // need to store extra ghost cells, and fixed place holder for some cells
    int total_rows = num_rows + 2 * ghost;
    int total_cols = num_cols + 2;

    // transferSize means the number of cells every subnode need to tranfer to the
    // main node to create the final graph
    int transferSize = (num_cols + 2) * num_rows;
```

```
// ghostSize means the numebr of cells every subnode need to tranfer between
// each other for every message passing along computation
int ghostSize = ghost * total_cols;

// since for a given number of ghost-cell size say g, a single node can get
// correct output for the cells for g iterations, so we only need to do
// iteraions/ghost number of message passing, and after each message passing
// we can iterate g times
int transfer = iterations / ghost; // assume always return an int

// allocate both blocks based on total_rows/cols
cells[0] = allocate_cells(total_cols, total_rows);
cells[1] = allocate_cells(total_cols, total_rows);

// allocate the final block only on master node, note that since each
// row have two extra fixed placeholder, it also need to be transfered
if (rank == 0) final_result = allocate_cells(real_cols + 2, real_rows);

// specify the current/next cell block
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, num_rows, ghost);

//conter for the iterations, used for determine if we need to
//create a output image based on iterations_per_snapshot
int current_iterations = 0;

for (int m = 0; m < transfer ; m++) {

    set(cells, num_cols, num_rows, ghost, rank, nodes);
    // The message passing tags is designed as below
    // four tags
    // 0 even to odd upward send (lower node to higher node)
    // 1 odd to even upward send
    // 2 even to odd downward send (higher node to lower node)
    // 3 odd to even downward send
    // in this way the message passing can be be finished(optimally) in 4 pass
    // for each tag, all the corresponding send/receive shall happen simultaniously
    // also note that the first/last nodes need to handle differently
    if (rank % 2 == 0) {
        // if it is the first node we only need even to odd upward send and odd to ↩
            even downward receive
        if (rank == 0) {
            MPI_Send(cells[cur_cells_index][num_rows], ghostSize, MPI_FLOAT, 1, 0, ↩
                MPI_COMM_WORLD);
            MPI_Recv(cells[cur_cells_index][num_rows + ghost], ghostSize, MPI_FLOAT, ↩
                1, 3, MPI_COMM_WORLD, NULL);
        }
        // if it is the last node and it is even
        // we only need odd to even upward receive and even to odd downward send
        else if (rank == nodes - 1) {
            MPI_Recv(cells[cur_cells_index][0], ghostSize, MPI_FLOAT, nodes - 2, 1, ↩
                MPI_COMM_WORLD, NULL);
            MPI_Send(cells[cur_cells_index][ghost], ghostSize, MPI_FLOAT, nodes - 2, ↩
                2, MPI_COMM_WORLD);
        }
        else {
            MPI_Send(cells[cur_cells_index][num_rows], ghostSize, MPI_FLOAT, rank + 1,↩
                 0, MPI_COMM_WORLD);
            MPI_Recv(cells[cur_cells_index][0], ghostSize, MPI_FLOAT, rank - 1, 1, ↩
                MPI_COMM_WORLD, NULL);
            MPI_Send(cells[cur_cells_index][ghost], ghostSize, MPI_FLOAT, rank - 1, 2,↩
                 MPI_COMM_WORLD);
            MPI_Recv(cells[cur_cells_index][num_rows + ghost], ghostSize, MPI_FLOAT, ↩
                rank + 1, 3, MPI_COMM_WORLD, NULL);
        }
    }
    else {
        // if it is the last node and it is odd
        // we only need even to odd upward receive and odd to even downward send
        if (rank == nodes - 1) {
            MPI_Recv(cells[cur_cells_index][0], ghostSize, MPI_FLOAT, nodes - 2, 0, ↩
                MPI_COMM_WORLD, NULL);
            MPI_Send(cells[cur_cells_index][ghost], ghostSize, MPI_FLOAT, nodes - 2, ↩
                3, MPI_COMM_WORLD);
        }
        else {
            MPI_Recv(cells[cur_cells_index][0], ghostSize, MPI_FLOAT, rank - 1, 0, ↩
                MPI_COMM_WORLD, NULL);
            MPI_Send(cells[cur_cells_index][num_rows], ghostSize, MPI_FLOAT, rank + 1,↩
                 1, MPI_COMM_WORLD);
            MPI_Recv(cells[cur_cells_index][num_rows + ghost], ghostSize, MPI_FLOAT, ↩
                rank + 1, 2, MPI_COMM_WORLD, NULL);
            MPI_Send(cells[cur_cells_index][ghost], ghostSize, MPI_FLOAT, rank - 1, 3,↩
                 MPI_COMM_WORLD);
        }
    }

    for (i = 0; i < ghost; i++) {
        // every time after calculation reset the boundary value and the hotspot
        set(cells, num_cols, num_rows, ghost, rank, nodes);
        // determine if we have a hotspot based on the size of the plate
        if ((real_cols > hotSpotCol) && (real_rows > hotSpotRow)) {
            // set the hotspot onto the correct node
            hotspot(cells[cur_cells_index], num_cols, num_rows, ghost, rank, nodes);
        }
        // Traverse the plate, computing the new value of each cell
        for (x = 1; x < total_rows - 1 ; x++) {
            for (y = 1; y < total_cols - 1; y++) {
                // iterate a calculation multiple times to increase the granularity
                for (iter = 0; iter < iters_per_cell; iter++) {
                //The new value of this cell is the average of the old values of this ↩
                    cell's four neighbors
                    cells[next_cells_index][x][y] = (cells[cur_cells_index][x - 1][y] ↩
                         +
                                                  cells[cur_cells_index][x + 1][y]  +
                                                  cells[cur_cells_index][x][y - 1]  +
                                                  cells[cur_cells_index][x][y + 1]) * ↩
                                                      0.25;
                }
            }
        }
        // increase the counter
        current_iterations += 1;

        // if current_iterations have reached the number of iterations to print a ↩
            snapshot
        // goto the create process
        if (current_iterations % iterations_per_snapshot == 0) goto createSnap;

        // if the process hasn't finished place to continue
        cont:

        // Swap the two arrays
        cur_cells_index = next_cells_index;
        next_cells_index = !cur_cells_index;
    }
}
// the process to create the snapshot based on current values calculated
createSnap:
//tag 10 for all the transfer
//transfer the all the calculated result to the node 0 (despite the ghost cells)
// node 0
if (rank == 0) {
    // transfer its own value to the front of the final map
    memcpy(final_result[0], cells[cur_cells_index][ghost], transferSize * sizeof(float↩
        ));
    // sequential receive all the data from node 1 to (nodes-1) and place them in the ↩
        right position
    for (int p = 1; p < nodes; p++) {
        MPI_Recv(final_result[p * num_rows], transferSize, MPI_FLOAT, p, 10, ↩
            MPI_COMM_WORLD, NULL);
    }
    // after message receiving, create the final snapshot
    create_snapshot(final_result, real_cols, real_rows, current_iterations);
}
// all other nodes
else {
    // send the main cell(exlude ghost cells) to node 0
    MPI_Send(cells[cur_cells_index][ghost], transferSize, MPI_FLOAT, 0, 10 , ↩
        MPI_COMM_WORLD);
}

// if the iteration hasn't finished go back into the iteration for-loop
if (current_iterations < (iterations - 1)) goto cont;
// finalze MPI
MPI_Finalize();
// Compute and output the execution time
time_t end_time = time(NULL);
printf("\nExecution time: %d seconds\n", (int) difftime(end_time, start_time));
return 0;
}

// set the hotspot
void hotspot(float **cells, int num_cols, int num_rows, int ghost, int rank, int nodes) {
// determine which node is the fixed cell in, and the exact position for that cell ↩
    regarding to a certain node
if ((hotSpotRow >= (rank * num_rows - ghost)) && (hotSpotRow <= ((rank + 1) * num_rows↩
     - ghost))) {
    cells[hotSpotRow - rank * num_rows + ghost][hotSpotCol] = 1000;
}
}

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

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

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

return array;
}
```

```c
// Sets all of the specified cells to their initial value.
// Assumes the existence of a one-cell thick boundary layer.
void initialize_cells(float **cells, int num_cols, int num_rows, int ghost) {
    int x, y;
    for (x = 0; x < num_rows + 2 * ghost; x++) {
        for (y = 0; y <= num_cols + 1; y++) {
            cells[x][y] = INITIAL_CELL_VALUE;
        }
    }
}

// set the fix boundary value (maybe fixed point as well)
void set(float ***cells, int num_cols, int num_rows, int ghost, int rank, int nodes) {
    int x, y, i;
    if (rank == 0) {
        for (x = ghost; x < num_cols + ghost; x++) cells[0][ghost - 1][x] = cells[1][ghost↩
             - 1][x] = TOP_BOUNDARY_VALUE;
    }
    if (rank == nodes - 1) {
        for (x = ghost; x < num_cols + ghost; x++) cells[0][num_rows + ghost][x] = cells↩
            [1][num_rows + ghost][x] = BOTTOM_BOUNDARY_VALUE;
    }
    for (y = 0; y <= num_rows+1; y++)
        cells[0][y][0] = cells[1][y][0] = LEFT_BOUNDARY_VALUE;
    for (y = 0; y <= num_rows+1; y++)
        cells[0][y][num_cols + 1] = cells[1][y][num_cols + 1] = RIGHT_BOUNDARY_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 num_cols, int num_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 (num_cols > 1000) {
        if ((num_cols % 1000) == 0) scale_y = num_cols / 1000;
        else {
            die("Cannot create snapshot for x-dimensions >1,000 that are not multiples of ↩
                1,000!\n");
            return;
        }
    }
    if (num_rows > 1000) {
        if ((num_rows % 1000) == 0) scale_x = num_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", num_cols / scale_x, num_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 (x = 0; x < num_rows; x += scale_x) {
        for (y = 0; y < num_cols; y += scale_y) {
            float sum = 0.0;
            for (i = x; i < x + scale_x; i++) {
                for (j = y; j < y + scale_y; j++) {
                    sum += cells[i][j];
                }
            }
            // Write out the average value of the cells we just visited
            int average = (int) (sum * inverse_cells_per_pixel);
            fprintf(out, "%d 0 %d\t", average, 100 - average);
        }
        fwrite("\n", sizeof(char), 1, out);
    }
    // Close the file
    fclose(out);
}

// directly print the value of the given pointer to the block and its row/column
// print only cell value instead of generate ppm file
// used only for debug process
void printMaps(float **cells, int num_cols, int num_rows, int ranks, int type) {
    char text[255];
    sprintf(text, "maps_in_%d_%d", ranks, type);
    FILE *out = fopen(text, "w");
    // Make sure the file was created
    if (out == NULL) {
        printf("Error creating snapshot file!\n");
        return;
    }
    int x, y;
    for (x = 0; x < num_rows; x++) {
        for (y = 0; y < num_cols; y++) {
            int val = (int)(cells[x][y]);
            fprintf(out, "%d ", val);
        }
        fwrite("\n", sizeof(char), 1, out);
    }
    fclose(out);
}

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

9.2 halo.slurm

```
#!/bin/bash
#SBATCH --nodes=20
#SBATCH --ntasks=200
#SBATCH --ntasks-per-node=10
#SBATCH --cpus-per-task=1
#SBATCH --partition=parallel
#SBATCH --account=parallelcomputing
#SBATCH --exclusive
#slurm script to run halo
#Jerry Sun (ys7va) 2017/4/12
module load openmpi
#compile with -O3 flag output executable as halo
mpicc -O3 -o halo halo.c
#run halo in 200 tasks
#condition: 10000 iterations on 10000 * 10000 plate, 1 iter per cell, output 1 final image↩
     using 2 ghost-cell
mpirun -np 200 halo 10000 10000 1 10000 10000 2
```