# Assignment2 Report Part1 Redblue

Name: Yahong Liu

SID: 440358628

## 1. Problem definition and Requirements

The Red/Blue computation simulates two interactive flows: an n by n board is initialized so cells have one of three colors: red, white, and blue, where white is empty, red moves right, and blue moves down. The board is initialized with 1/3 cells in read, 1/3 in white and 1/3 in blue and colors are interleaved and spread across the board. Colors wraparound to the opposite side when reaching the edge.

Viewing the board as overlaid with t by t tiles, so the grid is divided into t tiles in rows and columns. Each row or column of a tile has n/t squares.

In the first half step of an iteration, any red color can move right one cell if the cell to the right is white. On the second half step, any blue color can move down one cell if the cell below it is white. The case where red vacates a cell (first half step) and blue moves into it (second half step) is okay.

With the input threshold c, the computation terminates if any tile’s colored squares are more than c% one color (blue or red). Besides, with the input maximum iterations MAX\_ITRS, the computation terminates if the iterations have reached this boundary.

## 2. Parallel algorithm design

Firstly, with the input processor numbers, we need to separate the question into two parts: sequential and parallel.

Sequential program is the same with Assignment1, but we need to separate the tiles in 2-dimension for parallel program.

### 2.1 Sequential Program

For sequential computation, we need to create a dynamic two-dimensional array as the n by n grid. Then we need to allocate the color for each square with the 1/3 probability. We assign 1 to the square as red and 2 as blue. When computation started, red squares move in the first half step; When red movement finished in this round, blue squares start to move.

In each moving step, we assign 3 to the square when one colored square has just move in and assign 4 when the colored square has just move out. Using this method, we could ensure the colored squares are moving based on the original state in each round.

After movement, we need to count the colored squares and calculate the percentage in each tile separately. If any tile’s colored squares are more than c% one color, we assign the “finished” flag to TRUE and terminate the program.

### 2.2 Parallel Program

Parallel is similar to sequential program, but different in dealing with array.

Firstly, the largest processor number is so that we could separate the processors into two groups: normal group and obsolete group. Those processors whose rank is larger than must be blocked using MPI\_Barrier.

Secondly, inside the normal group, we will create a 2-dimensional Cartesian topology. As we have separated the processors, we have “numprocs\_group” of processors inside this normal group. So one dimension size is because this number must be the smallest size of the topology. Thus the other dimension is because this will maximize the use of group processors. With these two dimension sizes, we could create the Cartesian topology. In addition, this calculation of dimension size may leave some processors and we need to block these processors as well.

After creating the topology, we need to retrieve neighbors for each processor and calculate the width and height of each processor. With the calculations, we need four buffers for each processor: bufferUp, bufferDown, bufferLeft and bufferRight.

In first step of computation, we send maximum edge to bufferLeft of next processor, and get maximum edge of previous processor into its own bufferLeft. After that, we send minimum edge to bufferRight of previous processor, and get minimum edge of next proccessor into its own bufferRight. With the two buffers we could know the previous and next condition of edges. The red colored squares start to move based on the two buffers and the array in this processor.

In the second step of computation, we send maximum edge to bufferUp of next processor, and get maximum edge of previous processor into its own bufferUp. After that, we send minimum edge to bufferDown of previous processor, and get minimum edge of next proccessor into its own bufferDown. Again with the two buffers we could know the previous and next condition of edges. The blue colored squares start to move based on the two buffers and the array in this processor.

After movement, we need to count the colored squares and calculate the percentage in each tile separately. If any tile’s colored squares are more than c% one color, we assign the “finished” flag to TRUE and terminate the program in this processor.

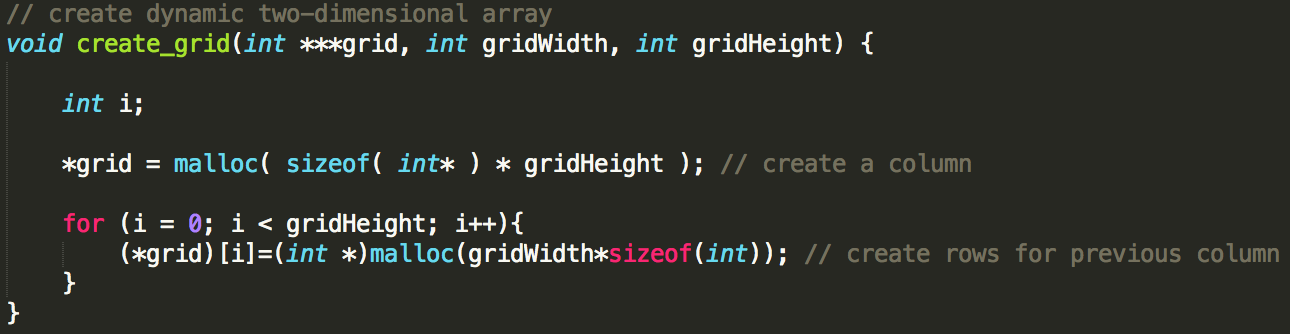
Then we need to broadcast all the “finished” flag and do the logical OR operation. Thus, if any processor has set “finished” flag to TRUE, the result of logical OR will be TRUE. Then we broadcast the “finished” flag to each processor to terminate the whole parallel program.

## 3. Implementation and Testing

### 3.1 Implementation

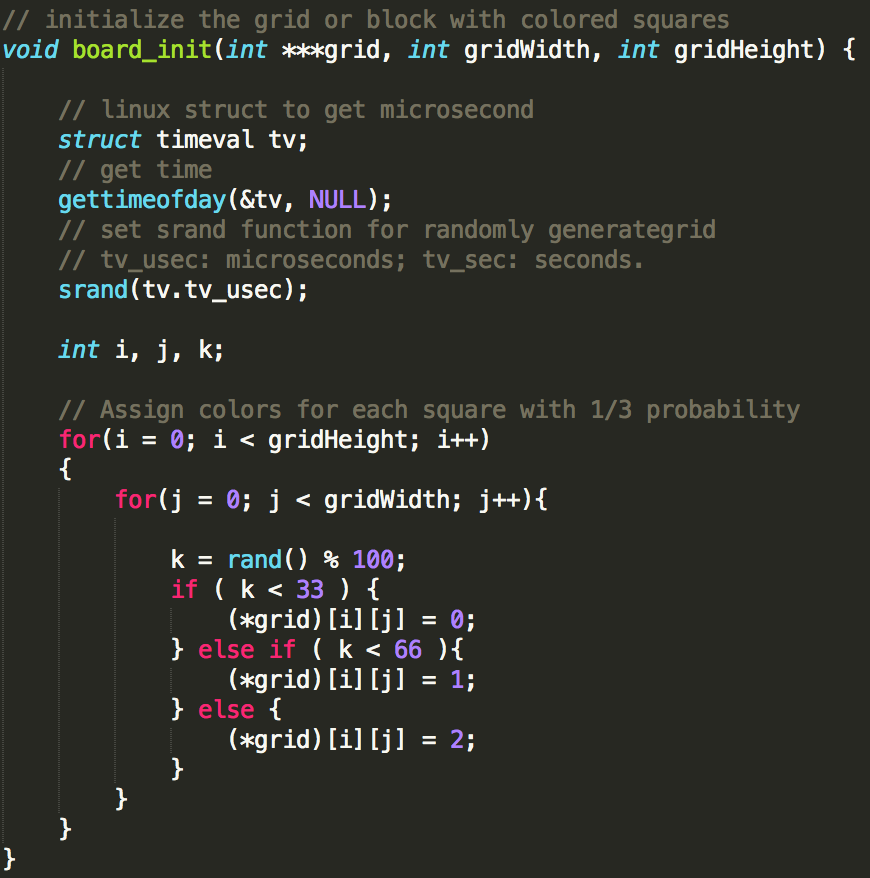
#### 3.1.1 Create dynamic two-dimensional array

Use malloc() to create the dynamic array based on the input grid size. This function could be used either sequential or parallel.



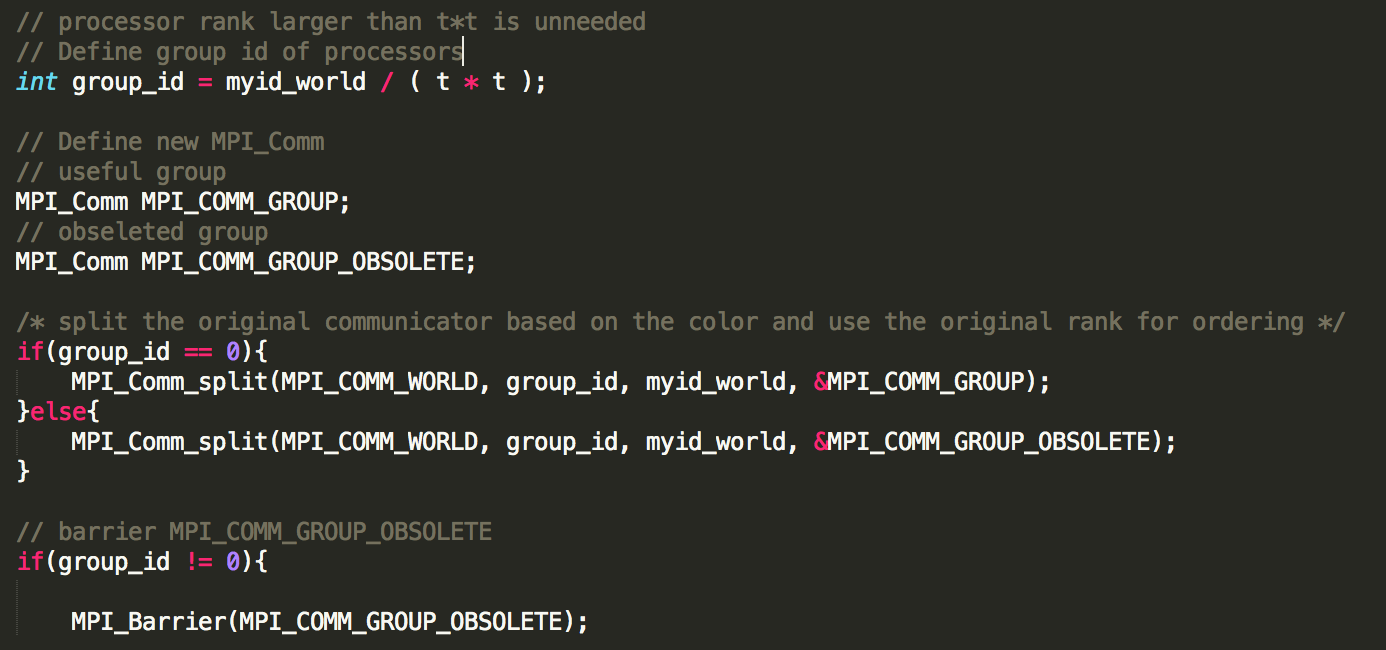
#### 3.1.2 Allocate the colored squares

I used the Linux function gettimeofday() to get the current seconds and microseconds. Then pass the microseconds to srand() function so that we could get different random numbers with function rand() in every microseconds. This function can be used either sequential or parallel.



#### 3.1.3 Split processors into different groups

Define the group id based on the tile amount. Then split each processor into groups. Block the unneeded group.



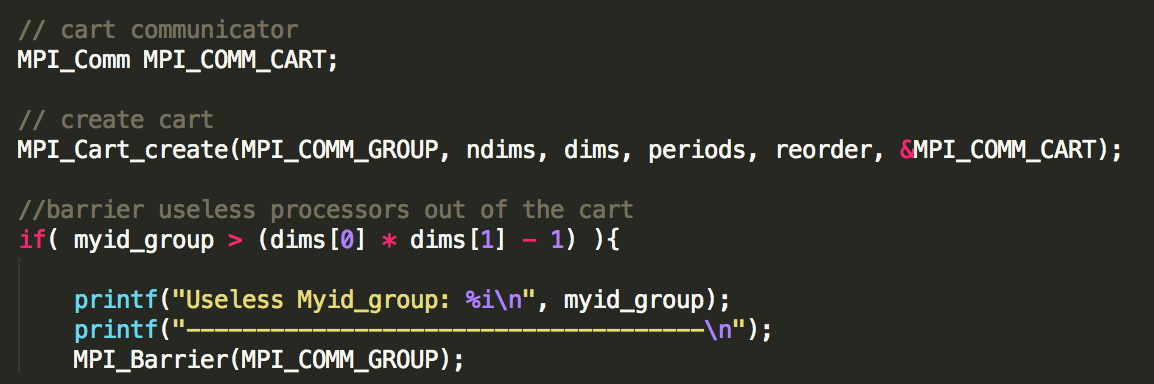
#### 3.1.4 Define the Cartesian topology

Define the size of dimension of Cartesian topology.



#### 3.1.5 Create the Cartesian topology

Create the topology and block the extra processors.



#### 3.1.6 Find neighbors and define processor size

Find neighbors in four directions and define the processor size.



To elaborate minimum and maximum edges, I take PROC\_SIZE\_UPDOWN as an example.

First I assume each processor has an extra tile row. This will get out of the range of grid. At this assumption step:

(1) Minimum edge is: ;

(2) Maximum edge is: .

However only if the processor meets the requirement: it will be allocating an extra tile row. Thus, we subtract the wrong allocation and will get the right boundary.

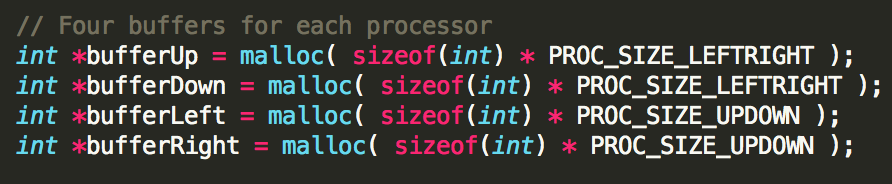
(1)

(2)

Thus the processor array size is . Similarly, we could get the PROC\_SIZE\_LEFTRIGHT.

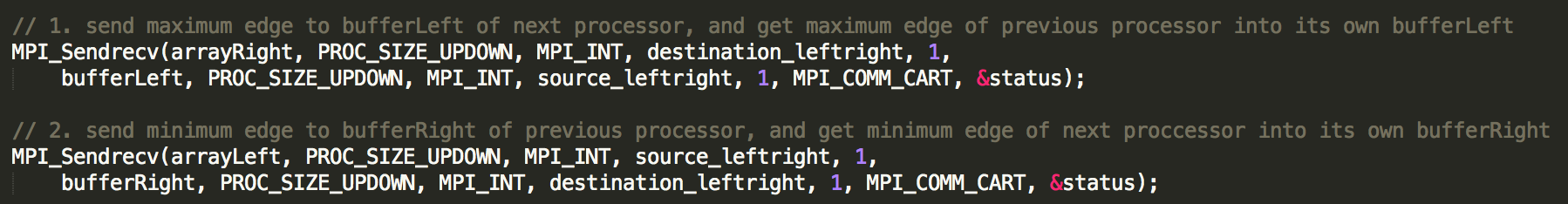
#### 3.1.7 Define buffers for sending and receiving edges

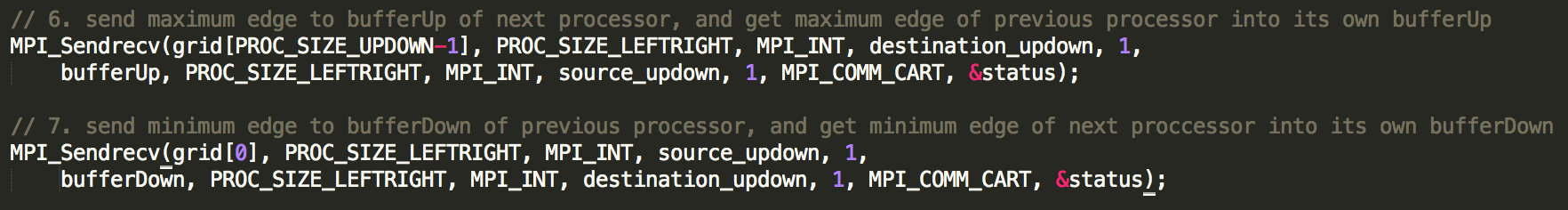
Define four buffers for each processor using previous processor sizes.



#### 3.1.8 Send and receive the edges

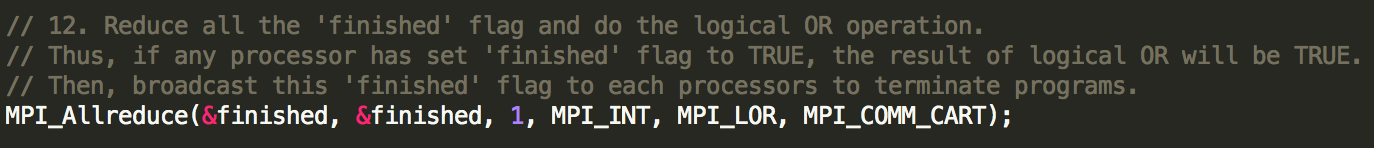
Send and receive the edges to four buffers.





#### 3.1.9 Reduce the “finished” flag and broadcast

Use MPI\_Allreduce() to reduce all the “finished” flag and do the logical OR operation, then broadcast the flag to each processor.



### 3.2 Testing

#### 3.2.1 Sequential

Use command to test sequential program.

## mpirun -np 1 redblue\_yahongliu 20 4 50 100

#### 3.2.2 Processors are smaller than t\*t

Use command to test parallel program.

## mpirun -np 5 redblue\_yahongliu 20 4 50 100

#### 3.2.3 Processors are equal to t\*t

Use command to test parallel program.

## mpirun -np 16 redblue\_yahongliu 20 4 50 100

#### 3.2.4 Processors are larger than t\*t

Use command to test parallel program.

## mpirun -np 20 redblue\_yahongliu 20 4 50 100

## 4. Manual

### 4.1 On Linux platform, extract tar file to current directory

## tar -xvf ./assignment2\_yahongliu.tar

### 4.2 Change the directory

## cd assignment2\_yahongliu/1\_redblue

### 4.3 Run Makefile to compile the program

## make

### 4.4 Run the program with processor number and parameters

Input the parameters: grid size, tile size, terminating threshold, and maximum number of iterations.

For example: 7 processors, grid size 20, tile size 4, threshold 50% and maximum iterations 100.

## mpirun -np 7 redblue\_yahongliu 20 4 50 100

### 4.5 Outputs

The outputs will be blue and red squares numbers and their percentage of each tile in every processor.

When any tile’s red or blue squares are more than the threshold, its processor will terminate. It will broadcast to other processors to terminate.