# Assignment1 Report

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.

### 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.

For different processor numbers, we will separate the grid in different ways:

(1) If tile number (t) is larger than processors and could be divisible by processors, we allocate tiles to each processor equally.

(2) If tile number (t) is larger than processors but cannot be divisible by processors, we first allocate tiles to each processor equally; The remaining tiles will allocate to the first few processors.

(3) If tile number (t) is smaller than processors, we split processors into groups and the group size is amount of tiles. In the first group, we allocate the tiles equally and start computation. For the remaining groups, we block the processors until the computation terminates.

After these allocation, each working processor will create their own two-dimensional array and allocate the colored squares. The processor width should be n and the processor height should be based on the previous allocation.

After then, we need two buffers in each processor: bufferUp and bufferDown. These two buffers are of size n.

In each round of computation, red squares move in the first half step. Then 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.

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.

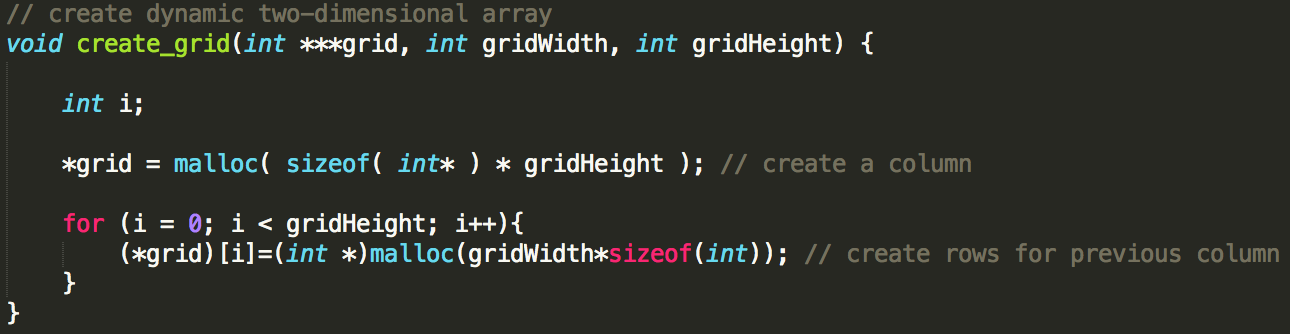
Then we need to reduce all the “finished” flag and do the logical OR operation. Thus, if any processor has set “finished” flag to TRUE, the result of logical OR will be TRUE. Then we broadcast the “finished” flag to each processor to terminate the whole parallel program.

## 3. Implementation and Testing

### 3.1 Implementation

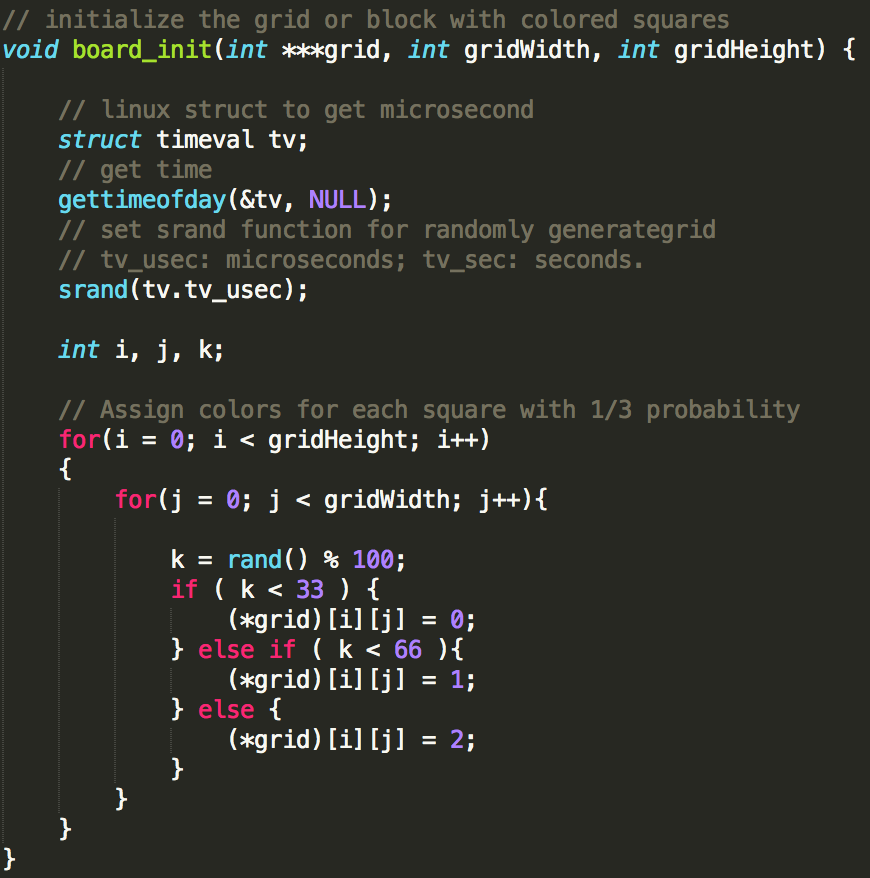
#### 3.1.1 Create dynamic two-dimensional array

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



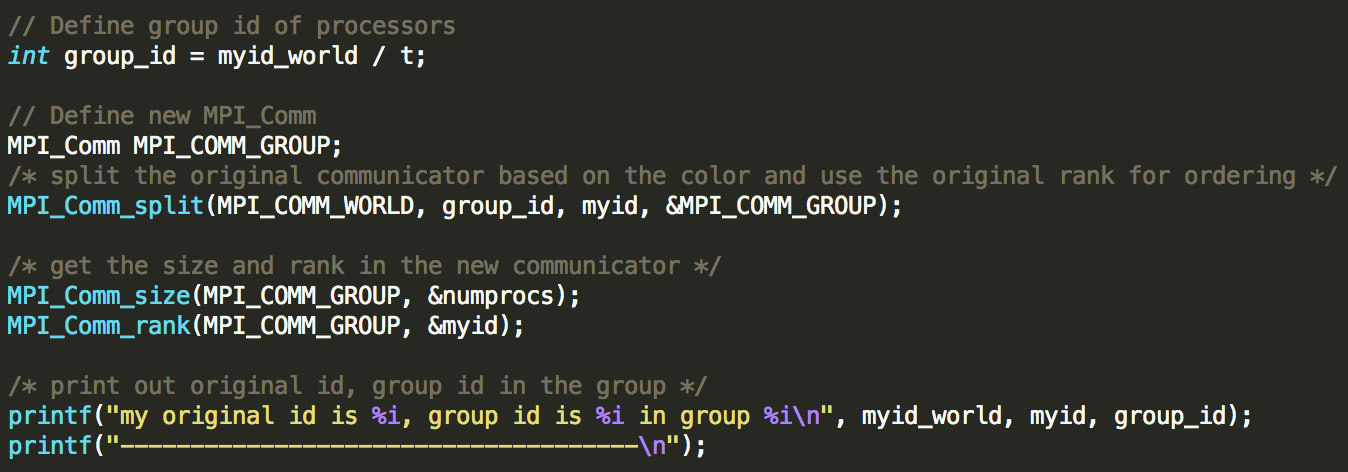
#### 3.1.2 Allocate the colored squares

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.

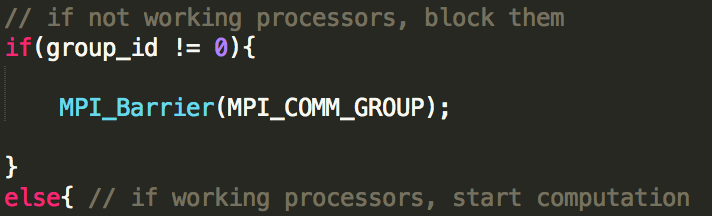


#### 3.1.3 Split processors into different groups

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



If this processor is not working processor, then block it until computation terminates.



#### 3.1.4 Define edges and size of each working processor

Define the size of each working processor.

For calculating minimum and maximum edges, 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: ;

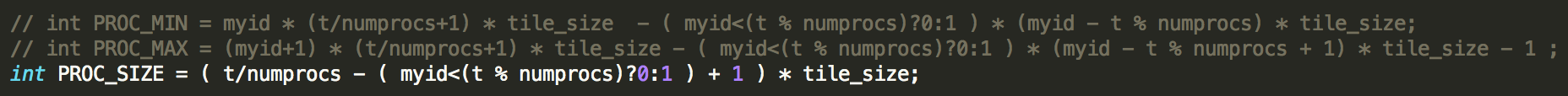
(2) Maximum edge is: .

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)

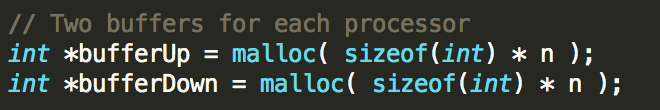
(2)

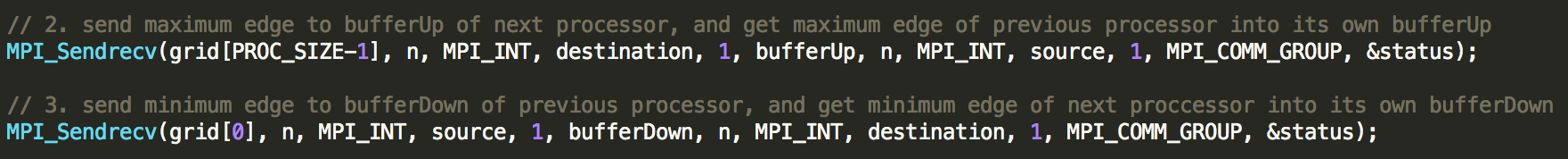
Thus the processor array size is .



#### 3.1.5 Define buffers and MPI communication

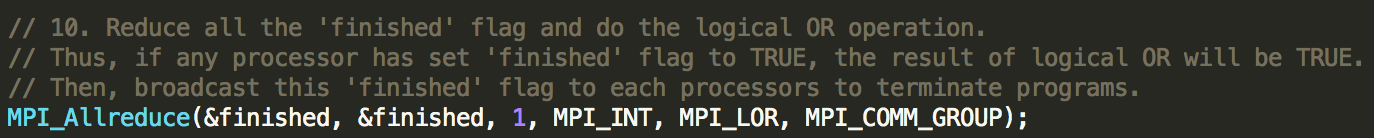
Define two buffers for using MPI\_Sendrecv() to communicate between the previous and next processor.





#### 3.1.6 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 150 15 50 100

#### 3.2.2 Tile number larger than processors and can be divisible

Use command to test parallel program.

## mpirun -np 10 redblue\_yahongliu 150 15 50 100

#### 3.2.3 Tile number larger than processors but cannot be divisible

Use command to test parallel program.

## mpirun -np 11 redblue\_yahongliu 150 15 50 100

#### 3.2.4 Tile number smaller than processors

Use command to test parallel program.

## mpirun -np 16 redblue\_yahongliu 150 15 50 100

## 4. Manual

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

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

### 4.2 Change the directory

## cd redblue\_yahongliu

### 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 number, terminating threshold, and maximum number of iterations.

For example: 7 processors, grid size 150, tile number 15, threshold 50% and maximum iterations 100.

## mpirun -np 7 redblue\_yahongliu 150 15 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.