The report of Red/Blue Computation

1. Problem definition and Requirements

   The Red/Blue Computation stimulates two interactive flows: an n*n grid is initialized by three colours, namely red, blue and white. Each colour take 1/3 of the total grid. Where white (0) is empty, red moves right and blue moves down. Colours wraparound to the opposite side when reaching the edge. In the first half step of an iteration, any red colour can move right one cell if the cell to the right is unoccupied (white); on the second half step, any blue colour can move down one cell if the cell below it is unoccupied (white); the case where red vacates a cell (first half) and blue moves into it (second half) is okay. Viewing the board as overlaid with t by t tiles (t divides n), the computation terminates if any tile's coloured squares are more than c% one colour (blue or red).

2. Parallel algorithm design

   The n by n board could be realized by a two-dimensional Arrays. Then, a variable called "block" could be defined by n divides t (for simplicity assume n is divisible by t). Depending on different total number of processes inputted by user, which means, "block" may be indivisible by the number of processes (this situation has been presented in Fig 1), and each process may have to handle different number of tiles. Thus, the definition of boundaries of each process would be important. In other words, the algorithm shall be able to assign the correct number of blocks to each process.

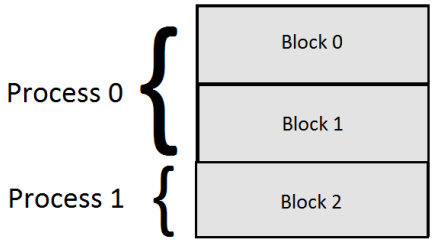

Fig 1 "block" indivisible by the number of processes

Two equations could solve the "boundary" problem;

$$Each\_proc\_block = \frac{block}{process}$$

$$Mod\_proc\_block = block \bmod process$$

If the process's rank less than *Mod_proc_block*, the quality of block needed to be operated by this process is (*Each_proc_block* +1); however, if the rank of process greater than the *Mod_proc_block*, this process need to handle *Each_proc_block* blocks.

Each process's "boundary" index could be calculated as follow ("myid" defined as the rank of process):

```
if (Mod_proc_block  != 0) {
    if (myid < Mod_proc_block ) {
        max_bondary = (myid + 1)*(Each_proc_block  + 1)*tile - 1;
        min_bondary = myid*(Each_proc_block  + 1)*tile;
    }
    else if (myid >= Mod_proc_block ) {
        min_bondary = myid * (Each_proc_block  + 1)*tile - (myid - Mod_proc_block)*tile;
        max_bondary = (myid + 1)*(Each_proc_block  + 1)*tile - 1 - (myid - Mod_proc_block + 1)*tile;
        }
    }
    else if (Mod_proc_block  == 0) {
        max_bondary = (myid + 1) * Each_proc_block  * tile - 1;
        min_bondary = myid * Each_proc_block  * tile;
}
```

For example, there are two processes (Process 0, Process 1) to be assigned to deal with this problem. After these two boundaries (Process 0's Max Boundary and Process 1's Min Boundary) array's index being calculated, the "Max Boundary" of the Process 0 could be able to compare its one-dimensional array with the "Min boundary" of the Process 1 to check whether the blue could move down. If the blue can move, both processes shall change their boundaries' array values. Thus, a number of processes could run at one time.

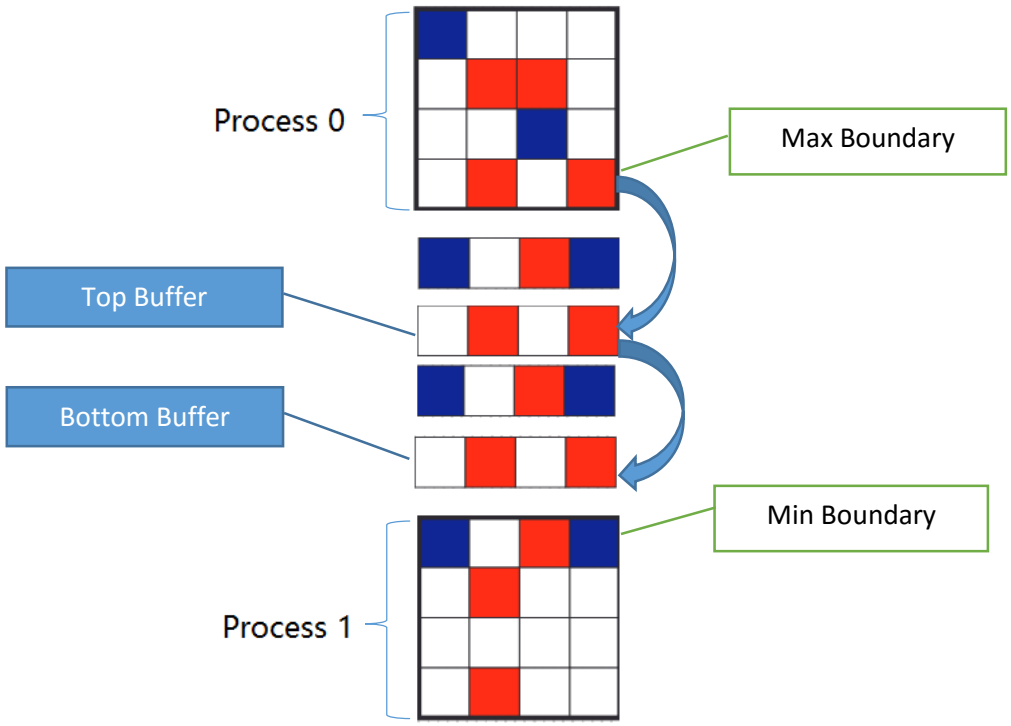

Fig 2. The example of comparing boundary array between two processes

3. Implementation and Testing

3.1. Implementation

The parallel algorithm could be realised by "MPI_Sendrecv" which has been defined below:

```
MPI_Sendrecv( void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest,
int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvtype, int source,
int recvtag, MPI_Comm comm, MPI_Status *status )
```

Besides, three buffers needed to be defined, namely "bufTop", "bufBut" and "gather_Buf". Generally, the "bufTop" stores the "Max Boundary" of the last process, and the "bufTop" will save the array of "Min Boundary" of the next process. The segment below presents how the process communicates with each other;

```
MPI_Sendrecv(grid[max_bondary], n, MPI_INT, destination, 1, bufTop,
n, MPI_INT, source, 1, MPI_COMM_WORLD, &status);
MPI_Sendrecv(grid[min_bondary], n, MPI_INT, source, 1, bufBut,
n, MPI_INT, destination, 1, MPI_COMM_WORLD, &status);
```

"MPI_Sendrecv" enables current process to send the message to the next process's buffer and receive the message from the last process. Thus, the boundaries' array could be exchanged by it. When the current process receives the message from the "bufTop", it will compares its array value in "Min Boundary" with the value in "bufTop", the implement procedures present as follow:

```
int y = 0;
for (y = 0; y < n; y++)
{
    if (bufTop[y] == 2 && grid[min_bondary][y] == 0) {
        bufTop[y] = 4;
        grid[min_bondary][y] = 3;
    }
}
```

Moreover, the same operation could be applied to compare the array value between “Max Boundary” and “bufBut”;

The final stage of the program is that when one process has a tile which has already fulfil the requirement, and this process has to notify the other processes that they need to stop running. Thus, two methods called MPI_Reduce and MPI_Bcast shall be employed to stop the other processes.

The code below shows that the value of “finished” in all the processes will be operated by “OR”, the result of “Or” will be sent to “process 0”’s buffer called “Allfinished”, then, the value of “Allfinished” will be broadcasted to other processes. The code below shows how to realize this function.

```
int allFinished;
MPI_Reduce(&finished, &allFinished, 1, MPI_INT, MPI_LOR, 0,MPI_COMM_WORLD);
finished = allFinished;
MPI_Bcast(&finished, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

When the number of processes exceed the maximum needing number (total block’s number), the function MPI_Comm_split could fix this problem. In other words, restricting the required (Maximum processes number) processes in the same communication domain, and using MPI_Barrier to stop extra processes. The code below could implement this function

```
int color = myid_world / block;
    MPI_Comm row_comm;
    MPI_Comm_split(MPI_COMM_WORLD, color, myid_world, &row_comm);
    MPI_Comm_rank(row_comm, &myid);
    MPI_Comm_size(row_comm, &numprocs);
    if(color != 0){
        MPI_Barrier( row_comm);
    }
```

3.2. Testing

3.2.1. Testing sequence program (one process)

The picture below presents the result of running by one process, the result is random, for the grid is filled by random algorithm.

```
-bash-4.2$ mpirun -np 1 r_1 100 10 50 50
Tiles[1][4]
Red count = 50
Blue count = 17
Red takes more than 50 % of the tile
Proc 0 already stopped, iteration: 12.
Red Percentage = 50.000000
Blue Percentage = 17.000000
```

Fig 3.2.1 Single process running result

3.2.2. Testing multi-processes

When the process number is more than one but not exceed the number of block, the running result has been presented below.

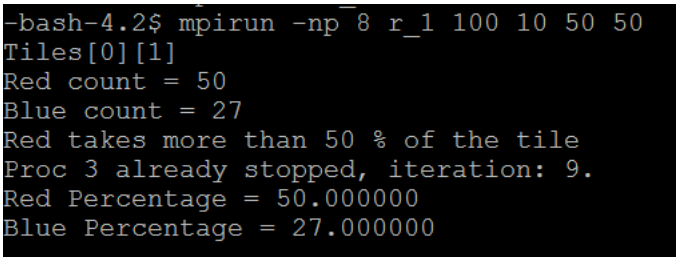

Fig 3.2.2 Multi-process running result

3.2.3. Testing maximum processes

In the example below, the value of block is 10. However the process number is 13, which exceeds the maximum number of needing processes. The system only run 10 of them, and barrier the extra 3 processes.

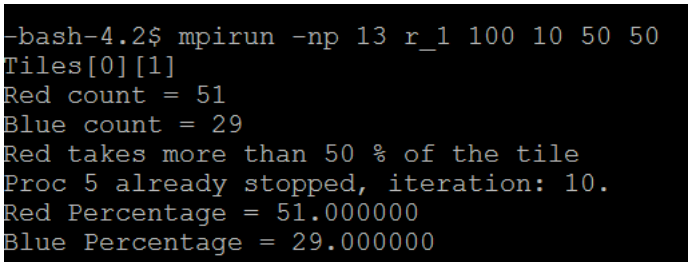

Fig 3.2.3 The result of the process's number exceeds the block number

4. Manual

After being compiling by the "*Makefile*" (by typing "make main"). To run it, user need to enter four arguments, namely grid size *n*, tile size *t*, terminating threshold *c*, and maximum number of iterations *max_iters*, like "*mpirun -np 5 redblue 16 4 60 70*", which means using five processes to run the program, the grid number is 16, tile size is 4, terminating threshold is 60%, and the maximum number of iterations is 70.