COMP5426

Assignment 1 – The red/blue computation

A. Problem definition and requirements:

In the red/blue computation, first we need to initialize a N by N size board. In this board, we have three different kinds of colours red, blue and white. We use 1 represent red, 2 represent blue and 0 represent white, we also need to make sure that there are roughly 1/3 cells in red, 1/3 cells in blue and 1/3 cells in white, they should be interleaved and spread across the board.

Second, the red can move right if its right cell is in white, the blue can move down if the cell below it is white, and colours can wraparound to the opposite side when reaching the edge. In one iteration, red and blue cells can only move one time, we will first move the cells in red, and then move the cells in blue.

Third, we need to separate the board into T by T tiles. As the board will perfectly overlaid with these tiles, each tile will have N/T by N/T cells, and the computation will be terminated if in any tiles, cells in one colour (red or blue) are more than the c% which is a input threshold value.

Forth, we need to have a counter to count how many iterations we have performed. If the number of iterations is larger than the predetermined maximum iterations, we can then terminate the computation without any tiles reach threshold value.

Last, we need to consider two situations which are sequential computation and parallel computation. In sequential computation we only have one process to perform the computation, and in parallel computation we need to consider that how to partition the board and use different processes to finish the computation.

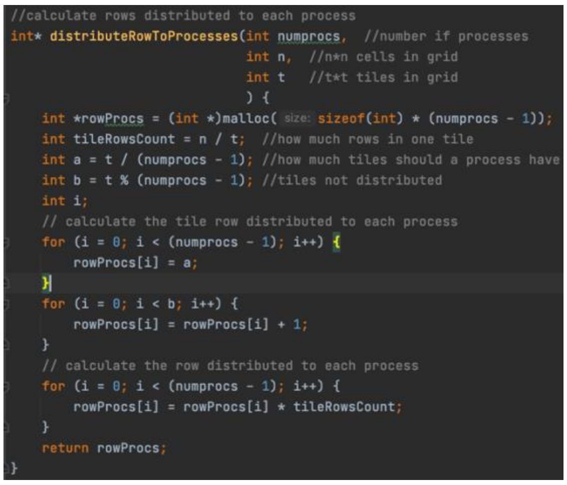
B. Parallel algorithm design

Based on the input value of processor, I need to divide the red&blue computation into two parts, one is sequential and the other is parallel.

The sequential design is that when the number of processor is one, the program performs sequential computation. The only processor should first create a board based on the input of n and t. Then performs red&blue computation, after every iteration it should check that whether in each tile there is one colour exceeds the threshold. If there is then the program terminates the computation and prints out the result, if there isn't just continues the computation until the iteration number reaches the maximum

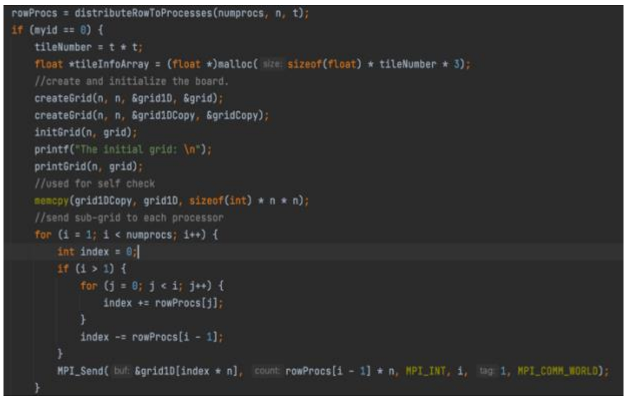
Then I will mainly focus on how I design the parallel algorithm. When the number of processor is larger than one, we need to move to parallel computation.

In my design, the processor 0 is the main processor. It is responsible for initializing the board distributing rows to each other processor and then send the corresponding data to other processors. But itself does not engage in parallel computation. I first use the number of T and the number of processors to calculate how much tile's rows should one processor get, then based on these to calculate how much rows should a processor get. The code is shown below.



```c
//calculate rows distributed to each process
int* distributeRowToProcesses(int numprocs,  //number if processes
                              int n,  //n*n cells in grid
                              int t   //t*t tiles in grid
                              ) {
    int *rowProcs = (int *)malloc( size: sizeof(int) * (numprocs - 1));
    int tileRowsCount = n / t;  //how much rows in one tile
    int a = t / (numprocs - 1); //how much tiles should a process have
    int b = t % (numprocs - 1); //tiles not distributed
    int i;
    // calculate the tile row distributed to each process
    for (i = 0; i < (numprocs - 1); i++) {
        rowProcs[i] = a;
    }
    for (i = 0; i < b; i++) {
        rowProcs[i] = rowProcs[i] + 1;
    }
    // calculate the row distributed to each process
    for (i = 0; i < (numprocs - 1); i++) {
        rowProcs[i] = rowProcs[i] * tileRowsCount;
    }
    return rowProcs;
}
```

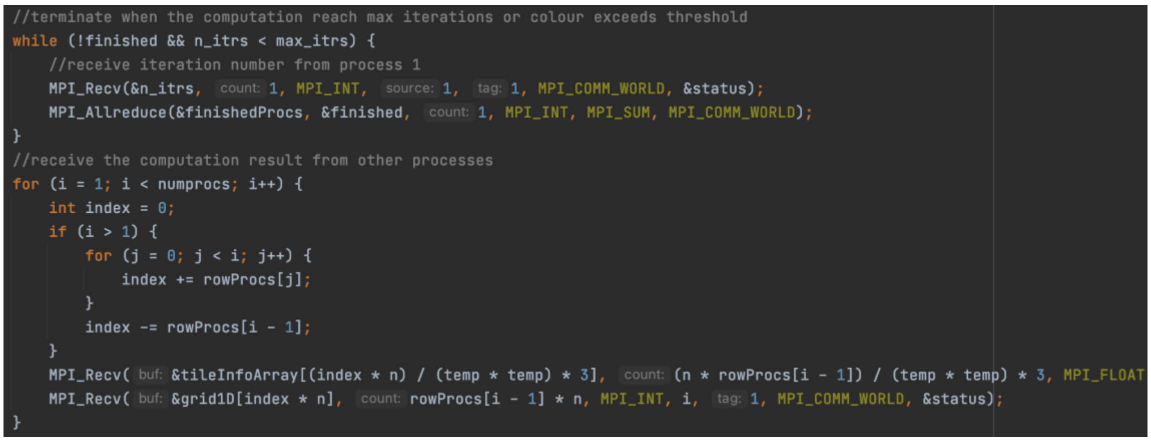
Fig 1



```c
rowProcs = distributeRowToProcesses(numprocs, n, t);
if (myid == 0) {
    tileNumber = t * t;
    float *tileInfoArray = (float *)malloc( size: sizeof(float) * tileNumber * 3);
    //create and initialize the board.
    createGrid(n, n, &grid1D, &grid);
    createGrid(n, n, &grid1DCopy, &gridCopy);
    initGrid(n, grid);
    printf("The initial grid: \n");
    printGrid(n, grid);
    //used for self check
    memcpy(grid1DCopy, grid1D, sizeof(int) * n * n);
    //send sub-grid to each processor
    for (i = 1; i < numprocs; i++) {
        int index = 0;
        if (i > 1) {
            for (j = 0; j < i; j++) {
                index += rowProcs[j];
            }
            index -= rowProcs[i - 1];
        }
        MPI_Send( buf: &grid1D[index * n],  count: rowProcs[i - 1] * n, MPI_INT, i,  tag: 1, MPI_COMM_WORLD);
    }
```

Fig 2

Then processor 0 act as a supervisor and monitor other processors, all other processors go to the red&blue computation stage. They use the data they have been assigned to perform computation. At last, when the computation should be terminated all other processors will send their own sub-grids and the computation results to the processor 0. The processor 0 will print out the final grid and result, also perform the sequential computation for a self-checking. The sample code is shown as Fig 3.



```c
//terminate when the computation reach max iterations or colour exceeds threshold
while (!finished && n_itrs < max_itrs) {
    //receive iteration number from process 1
    MPI_Recv(&n_itrs,  count: 1, MPI_INT,  source: 1,  tag: 1, MPI_COMM_WORLD, &status);
    MPI_Allreduce(&finishedProcs, &finished,  count: 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
}
//receive the computation result from other processes
for (i = 1; i < numprocs; i++) {
    int index = 0;
    if (i > 1) {
        for (j = 0; j < i; j++) {
            index += rowProcs[j];
        }
        index -= rowProcs[i - 1];
    }
    MPI_Recv( buf: &tileInfoArray[(index * n) / (temp * temp) * 3],  count: (n * rowProcs[i - 1]) / (temp * temp) * 3, MPI_FLOAT
    MPI_Recv( buf: &grid1D[index * n],  count: rowProcs[i - 1] * n, MPI_INT, i,  tag: 1, MPI_COMM_WORLD, &status);
}
```

Fig 3

All processors are responsible for performing red&blue computation of their own sub-grid. There is one thing that needed to considered is that blue can move vertically, so if we only consider the own sub-grid that each processor has been distributed, the final result will be wrong. Based on this we need to create ghost line for each sub-grid. How I create ghost line for each sub-grid is shown as Fig 4 and Fig 5. I will briefly explain this. Suppose we have a 8*8 grid, 2*2 tile, and 2 working processor. For each processor, it will get 2 tiles a 4*8 sub-grid. I add 2 rows for each sub-grid, so each processor will get a 6*8 sub-grid. The data in ghost row is from the real row, like in grey line 1, it is the same as the white line 1. Supposed Fig 4 represents gird 1 and Fig 5 represents gird 2, gird 1 should use grid 2's last line as its first ghost line, use grid 2's first line as its last ghost line. Every iteration, these two processors will send related ghost row information to each other so they can know the situation of their edges. Based on this each processor can know the data information at the edge to ensure correctness of the final computation result. The code is show in

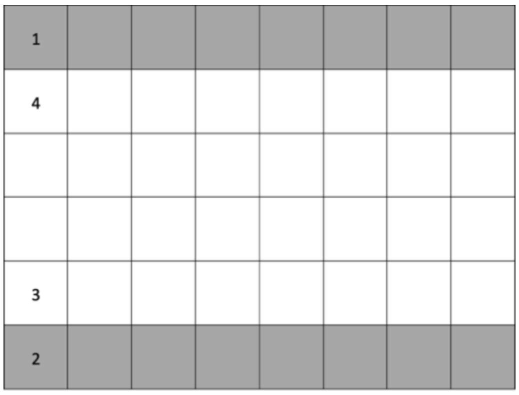


Fig 4.

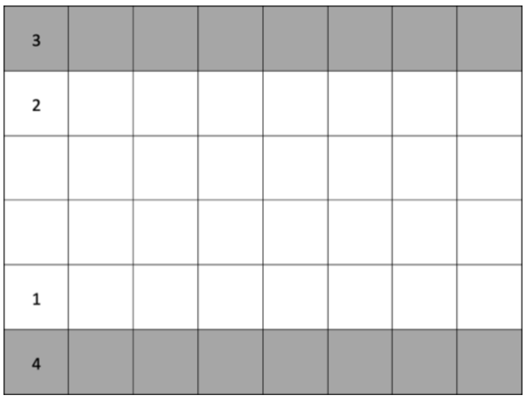
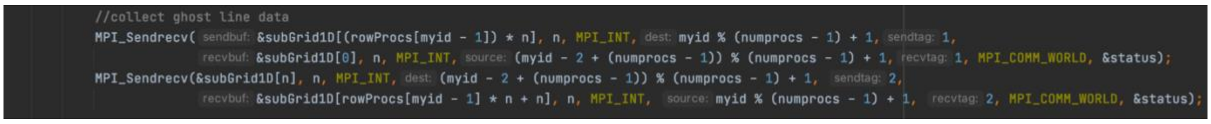


Fig 5.



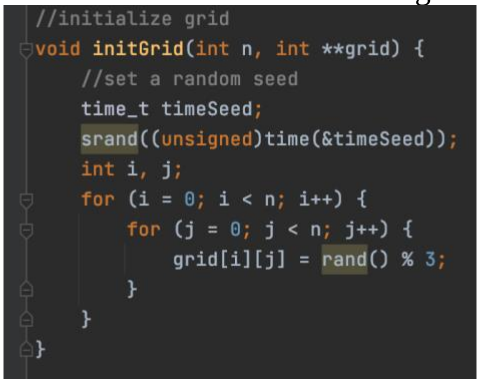
```
//collect ghost line data
MPI_Sendrecv( sendbuf: &subGrid1D[(rowProcs[myid - 1]) * n], n, MPI_INT, dest: myid % (numprocs - 1) + 1, sendtag: 1,
              recvbuf: &subGrid1D[0], n, MPI_INT, source: (myid - 2 + (numprocs - 1)) % (numprocs - 1) + 1, recvtag: 1, MPI_COMM_WORLD, &status);
MPI_Sendrecv(&subGrid1D[n], n, MPI_INT, dest: (myid - 2 + (numprocs - 1)) % (numprocs - 1) + 1,  sendtag: 2,
              recvbuf: &subGrid1D[rowProcs[myid - 1] * n + n], n, MPI_INT,  source: myid % (numprocs - 1) + 1,  recvtag: 2, MPI_COMM_WORLD, &status);
```

Fig 6.

C. Implementation and Testing

C.1 Implementation:

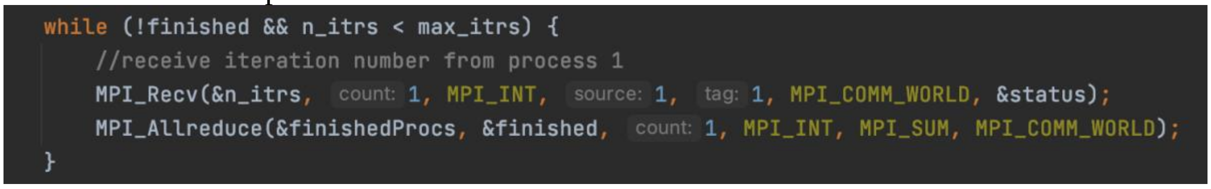
Initialize the whole grid: first use system time to get a seed and use this seed to generate a random number to initialize the grid.



```
//initialize grid
void initGrid(int n, int **grid) {
    //set a random seed
    time_t timeSeed;
    srand((unsigned)time(&timeSeed));
    int i, j;
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            grid[i][j] = rand() % 3;
        }
    }
}
```

Fig 7.

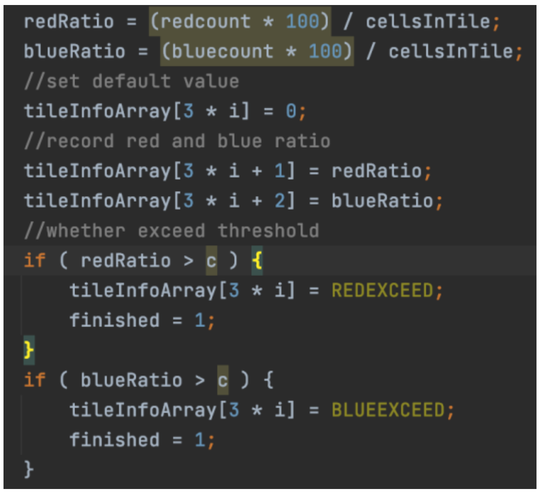
The processor 0 monitors other processors: it receives number of iterations and finished situation from other processors.



```
while (!finished && n_itrs < max_itrs) {
    //receive iteration number from process 1
    MPI_Recv(&n_itrs,  count: 1, MPI_INT,  source: 1,  tag: 1, MPI_COMM_WORLD, &status);
    MPI_Allreduce(&finishedProcs, &finished,  count: 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
}
```

Fig 8.

Check each tile’s information: use an array to store each tile’s information, the array’s length equals to three multiply all tile numbers, as we have T*T tile numbers, the length will be 3*T*T. 3*T will record whether this tile exceeds threshold and what colour exceeds the threshold, 3*T+1 will record the information about red ratio, 3*T+2 will record information about blue ratio.

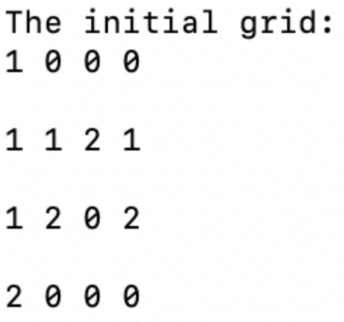


```
redRatio = (redcount * 100) / cellsInTile;
blueRatio = (bluecount * 100) / cellsInTile;
//set default value
tileInfoArray[3 * i] = 0;
//record red and blue ratio
tileInfoArray[3 * i + 1] = redRatio;
tileInfoArray[3 * i + 2] = blueRatio;
//whether exceed threshold
if ( redRatio > c ) {
    tileInfoArray[3 * i] = REDEXCEED;
    finished = 1;
}
if ( blueRatio > c ) {
    tileInfoArray[3 * i] = BLUEEXCEED;
    finished = 1;
}
```

Fig 9.

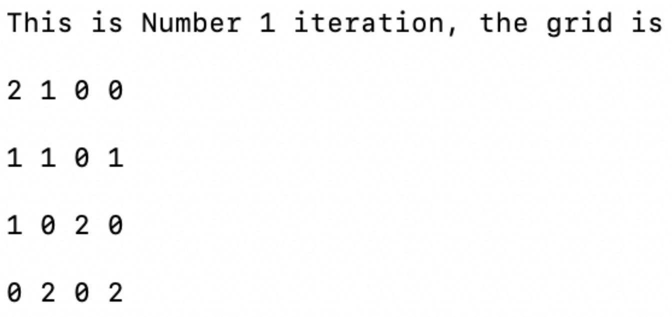
C.2 Testing
I perform serval tests to test my program.
1.Sequential
Two command to test sequential computation.
(1) mpirun -np 1 a1_ls 30 10 70 10
(2) mpirun -np 1 a1_ls 4 2 55 10
2.Parallel
Three command to test parallel computation.
(1) mpirun -np 3 a1_ls 30 10 70 10
(2) mpirun -np 4 a1_ls 30 10 70 10
(3) mpirun -np 3 a1_ls 100 20 70 10

I just use mpirun -np 1 a1_ls 4 2 55 10 to test. I use the initial grid, and the grid printed each iteration to check one by one to make sure red and blue are moving as required. The result is shown as below.



```
The initial grid:
1 0 0 0

1 1 2 1

1 2 0 2

2 0 0 0
```

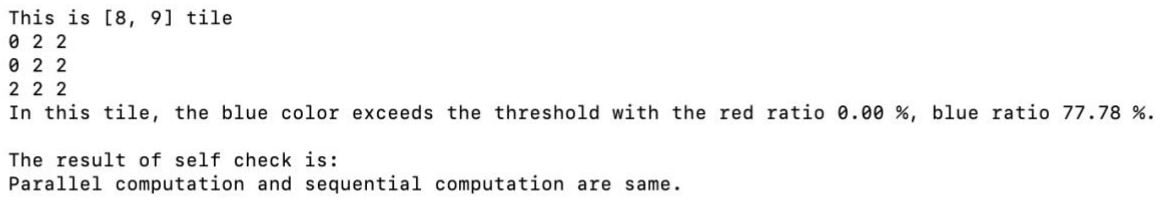
Fig 10.



```
This is Number 1 iteration, the grid is

2 1 0 0

1 1 0 1

1 0 2 0

0 2 0 2
```

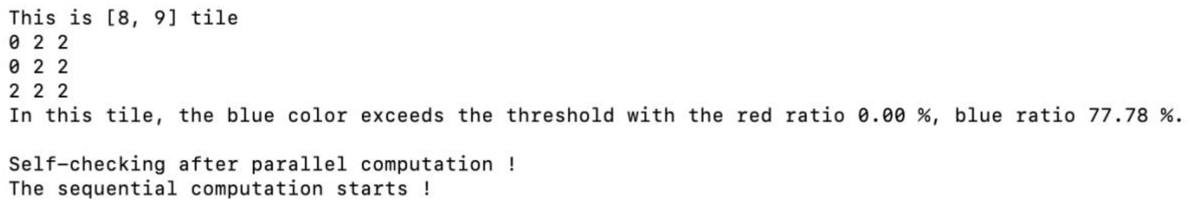
Fig 11.

For parallel computation, I just based on the record for which tile exceeds the threshold and the final check, if tile information results recorded for parallel is the same as the sequential, and the final self-check is also same, then I can ensure the parallel computation is also true. The test case I run is mpirun -np 4 a1_ls 30 10 70 10, the results are as below. Fig 12 is the result for parallel computation, Fig 13 is for sequential computation and self-check.



```
This is [8, 9] tile
0 2 2
0 2 2
2 2 2
In this tile, the blue color exceeds the threshold with the red ratio 0.00 %, blue ratio 77.78 %.

The result of self check is:
Parallel computation and sequential computation are same.
```

Fig 12.



```
This is [8, 9] tile
0 2 2
0 2 2
2 2 2
In this tile, the blue color exceeds the threshold with the red ratio 0.00 %, blue ratio 77.78 %.

Self-checking after parallel computation !
The sequential computation starts !
```

Fig 13.

D. Possible improvement

The major improvement I think I should implement in my program is that let process 0 engage in the parallel computation, because I lose one processor in the parallel computation and it has a bad influence on the performance. I believe that can be an improvement.

E. Manual

The example steps to run the program.
1. unzip the a1_ls.zip file
2. cd a1_ls (Make sure change to the directory contains file a1_ls.c)
3. make
4. mpirun -np 4 a1_ls 30 10 70 10
5. After that use "make clean" to clean the project

Input:

The 4 inputs we need to input are the n which represents n*n grid, t which represents t*t tile, c which represent threshold, and maximum iterations. Besides we can use the number after np to control the number of processors

Output:

When the number of processors is one we will run sequential computation, we can see the results of each iteration printed. If there is any tiles exceed the threshold, we can also see which tiles exceed and the detailed information in those tiles. If the number of processors is larger than we, the output will have the results of parallel computation, and followed by a self-check sequential computation.