Assignment1 The Red/Blue computation

Yang Ge SID:450005028

Problem definition and Requirements

This assignment is to do cell movements in a n by n grid, which is initialised with roughly 1/3 cells in red, 1/3 cells in blue and 1/3 cells in white, and use parallel algorithm.

Red cell can move to its right neighbour if its right neighbour is white, leftmost red cell can move to rightmost cell if corresponding rightmost cell is white before entire movement, blue cell can to its lower neighbour if its lower neighbour is white, and blue cell in last row can move to top row if the corresponding cell in top row is white before entire movement. The movements would start from left to right, then top to bottom. The grid can be perfectly overlaid with a t by t grid of tiles (i.e., t divides n, and every tile contains n/t by n/t cells). The computation terminates if any tile's coloured cells(blue or red) are more than c% or it achieve the maximum number of iterations .

- Use MPI to write a solution to this assignment
- Assume the processes are organised as a one-dimensional linear array
- Each process will hold a number of k rows of tiles, or k*n /rows of cells for k>= 0
- For the purpose of load balancing, processes should have roughly the same number of rows and the difference not greater than one tile row, or n/t cell rows
- Program must produce correct results for npros being greater than or equal to one
- Program needs to ask for 4 user defined parameters(integers) as inputs: cell grid size n, tile grid size t, terminating threshold c, and maximum number of iterations max_iters.
- Program needs to print out which tile has the coloured squares more than c% one colour(blue or red)
- After the parallel computation, main program must conduct a self-checking. i.e. first perform a sequential computation using the same data set and then compare the two results.

Parallel algorithm design

Master processor(processor 0) first create a random n*n grid.

For number of processor is equal to 1, processor 0 do a sequential computing.

Otherwise, processor 0 divide the initial grid and send sub-grids(tile(s)) to other processors, other processors receive data and make their own grid.

Before movement, every processor need to communicate with its neighbour processors(next processor and previous processor) to get the data of the row before its first row and the row after its last row for blue cell movement, then every processor can do red ,blue movement, check if finished in parallel, then use MPI_Allreduce to check if there is any finished processor, if so, terminate the computing. Repeat these steps until any tile achieve the requirement or computing achieve the maximum number of iterations. After terminating, all processors use MPI_Gatherv to send their results to processors 0. Processor 0 then do sequential computing and compare with parallel computing result.

There may be some processors which do not get grid from processor 0 when number of processor is greater than t, these processors just use MPI_Allreduce and MPI_Gatherv(use any number that do not affect other processors' results) to make MPI_Allreduce and MPI_Gatherv work.

Implementation and Testing

Using C language and MPI.
Processor 0 first create a random grid. The idea is make an 1 dimensional array with size n*n, shuffle it and change it to 2 dimensional array.(function randomize() is from https://www.geeksforgeeks.org/shuffle-a-given-array/)

```
void board_init(int cell_size, int grid[cell_size][cell_size]) {
    int arraysize = cell_size*cell_size;
    int tmp[arraysize];
    int i, j;
    for (i = 0; i < arraysize; i++) {
        if (i < arraysize / 3) {
            tmp[i] = 1;
        }
        else if (i < 2 * arraysize / 3) {
            tmp[i] = 2;
        }
        else {
            tmp[i] = 0;
        }
    }
    randomize(tmp, arraysize);
    int count = 0;
    for (i = 0; i < cell_size; i++) {
        for (j = 0; j < cell_size; j++) {
            grid[i][j] = tmp[count];
            count++;
        }
    }
}
```

Divide initial gird, and send sub-grids to other processors(number of processor > t, and number of processor < t)

```
if (t % numprocs != 0) {
    current_row += tile_size*(t / numprocs + 1);
}
else {
    current_row += tile_size * (t / numprocs);
}
num_of_row_in_first_processor = current_row;
rows_numbers[0] = num_of_row_in_first_processor;

for (m = 1; m < numprocs; m++) {
    //fisrt set num of tile row for each processors
    if ((t % numprocs != 0) && ((t % numprocs) >  m)) {
        num_t_row_in_processor = t / numprocs + 1;
    }
    else {
        num_t_row_in_processor = t / numprocs;
    }
    // send size information
    MPI_Send(&num_t_row_in_processor, 1, MPI_INT, m, 1, MPI_COMM_WORLD);

    rows_numbers[m] = tile_size*num_t_row_in_processor;

    //make 1D array for sending data
    int sending[num_t_row_in_processor * tile_size * n];
    sending_count = 0;
    //set sending array
    for (i = 0; i < (num_t_row_in_processor * tile_size); i++) {
        for (j = 0; j < n; j++) {
            sending[sending_count] = grid[current_row][j];
            sending_count++;
        }
        current_row++;
    }
    //send this 1D array
    MPI_Send(sending, (num_t_row_in_processor * tile_size * n), MPI_INT,
        MPI_COMM_WORLD);
    //all slaves get their rows
}
```

```
else if (numprocs > t) {
    //each working processor have tile row 1
    num_of_row_in_first_processor = tile_size;
    current_row = tile_size;
    num_t_row_in_processor = 1;
    rows_numbers[0] = tile_size;
    for (m = 1; m < numprocs; m++) {
        //e.g. processor 0, 1, 2, 3, 4, 5,  num of tile rows is 4, first 4 processors
            can get job, then
        //when it comes to the fifth processor(have id 4) and after this processor,
            cannot get job
        if (m == t) {
            //0 means no job
            num_t_row_in_processor = 0;
        }
        //save work load information
        rows_numbers[m] = num_t_row_in_processor * tile_size;
        //send work load(tile row number) to other processors
        MPI_Send(&num_t_row_in_processor, 1, MPI_INT, m, 1, MPI_COMM_WORLD);
        //then send actual row to working processors
        if (num_t_row_in_processor != 0) {
            int sending[num_t_row_in_processor * tile_size * n];
            sending_count = 0;
            for (i = 0; i < (num_t_row_in_processor * tile_size); i++) {
                for (j = 0; j < n; j++) {
                    sending[sending_count] = grid[current_row][j];
                    sending_count++;
                }
                current_row++;
            }
            //send
            MPI_Send(sending, (num_t_row_in_processor * tile_size * n), MPI_INT, m, 2,
                MPI_COMM_WORLD);
        }
    }
```

Communication, Move and Checking in Processor 0

```
//actual last row
int mylast[n];
int mylast_counter = 0;
//actual first row
int myfirst[n];
int myfirst_counter = 0;
//virtual first row
int topghost[n];
//virtual last row
int bottomghost[n];
//overall 2D array(contains real and virtual rows)
int sub_grid[num_of_row_in_first_processor + 2][n];
int sub_grid_counter = 0;

//set real rows
int row_c = 0, col_c = 0;
for (row_c = 1; row_c <= num_of_row_in_first_processor; row_c++) {
    for (col_c = 0; col_c < n; col_c++) {
        int tocopy = grid[row_c - 1][col_c];
        sub_grid[row_c][col_c] = tocopy;
    }
}
```

```
current_count = 0;

while (max_allreduce_variable == 0 && current_count < max_iters) {
    //get real last row
    for (mylast_counter = 0; mylast_counter < n; mylast_counter++) {

        mylast[mylast_counter] = sub_grid[num_of_row_in_first_processor]
            [mylast_counter];
    }

    //get real first row
    for (myfirst_counter = 0; myfirst_counter < n; myfirst_counter++) {
        myfirst[myfirst_counter] = sub_grid[1][myfirst_counter];
    }

    int real_last_processor = 0;
    //set real last processor
    if (numprocs > t) {
        //e.g. numprocs = 5 (0,1,2,3,4), n_t_r = 4, then the real last one would be
            processor(myid) 3
        real_last_processor = t - 1;
    }
    else {
        real_last_processor = (myid - 1 + numprocs) % numprocs;
    }
    //communicate with second row and real last row(send real last row to second row,
        receive top ghost from real last row)
    MPI_Sendrecv(mylast, n, MPI_INT, (myid + 1 + numprocs) % numprocs, 1, topghost, n,
        MPI_INT, real_last_processor, 1, MPI_COMM_WORLD, &status);
    //send real first row to real last row, receive bottom ghostt from second row
    MPI_Sendrecv(myfirst, n, MPI_INT, real_last_processor, 1, bottomghost, n, MPI_INT,
        (myid + 1 + numprocs) % numprocs, 1, MPI_COMM_WORLD, &status);

    //set top and bottom ghost in 2D array
    for (sub_grid_counter = 0; sub_grid_counter < n; sub_grid_counter++) {
        sub_grid[0][sub_grid_counter] = topghost[sub_grid_counter];
        sub_grid[num_of_row_in_first_processor + 1][sub_grid_counter] =
            bottomghost[sub_grid_counter];
    }
    //red movement
    red_movement(num_of_row_in_first_processor + 2, n, sub_grid);
    ////then blue movement
    blue_movement(num_of_row_in_first_processor + 2, n, sub_grid);
    //increase current count, will compare with max_iters in the beginning of the loop
    current_count++;
    //if finished in processor 0
    if (finished(n, num_of_row_in_first_processor + 2, sub_grid, tile_size, c, 1, 0,
        0) == 1) {
        allreduce_variable = 1;
    }
    // collect all if finished data
    MPI_Allreduce(&allreduce_variable, &max_allreduce_variable, 1, MPI_INT, MPI_MAX,
        MPI_COMM_WORLD);
    //if finished, break the loop, the movement part is over, do self-checking part
    if (max_allreduce_variable == 1) {
        break;
    }

}
```

Other processors first receive the data from processor 0, then do similar job(some processors who do not have a job may just use MPI_Allreduce and wait until entire movement finish)

Red movement

```
void red_movement(int r, int c, int gridr[r][c]) {
    int i, j;
    int n = c;
    for (i = 0; i < r; i++) {

        if (gridr[i][0] == 1 && gridr[i][1] == 0) {
            gridr[i][0] = 4;
            gridr[i][1] = 3;
        }
        for (j = 1; j < c; j++) {
            if (gridr[i][j] == 1 && (gridr[i][(j + 1) % n] == 0)) {
                gridr[i][j] = 0;
                gridr[i][(j + 1) % n] = 3;
            }
            else if (gridr[i][j] == 3) {
                gridr[i][j] = 1;
            }
        }

        if (gridr[i][0] == 3) {
            gridr[i][0] = 1;
        }
        else if (gridr[i][0] == 4) {
            gridr[i][0] = 0;
        }
    }
}
```

Blue movement in sequential computing is similar to Red movement. Blue movement in other processors is slightly different because in other processors, blue cells in last row cannot move to top row, while they can in sequential computing, because in other processors the grid is just a part of the initial grid.

For testing, all processors use MPI_Gatherv(), and processor 0 can get all parallel results from other processors, then compare the overall result with sequential result, check if they are matched.

```
sendbuf = (int *)malloc(num_of_row_in_first_processor*n * sizeof(int));
int sendbuf_count = 0;
sendcount = num_of_row_in_first_processor*n;
for (i = 1; i <= num_of_row_in_first_processor; i++) {
    for (j = 0; j < n; j++) {
        sendbuf[sendbuf_count] = sub_grid[i][j];
        //printf("%d\n", sendbuf[sendbuf_count]);
        sendbuf_count++;
    }
}
//recvcounts, displs
int current_displs = 0;
for (i = 0; i < numprocs; i++) {
    displs[i] = current_displs;
    current_displs += rows_numbers[i]*n;
    recvcounts[i] = rows_numbers[i] * n;
}

MPI_Gatherv(sendbuf, sendcount, MPI_INT, recvbuf, recvcounts, displs, MPI_INT, 0,
    MPI_COMM_WORLD);
//self_checking
sequential_computation(n, grid, tile_size, c, max_iters, 1);
int para_result[n][n];
int compare_flag = 0;
int compare_count = 0;

for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        para_result[i][j] = recvbuf[compare_count];
        compare_count++;
    }
}
```

```
    for (i = 0; i < n; i ++) {
        for (j = 0; j < n; j++) {
            if (para_result[i][j] != grid[i][j]) {
                compare_flag = 1;
                printf("error! in row number %d, column number %d, sequential result is
                    %d, parallel result is %d\n", i, j, grid[i][j], para_result[i][j]);
            }

        }
    }
    if (compare_flag == 0) {
        printf("campared with sequential result, all good!\n");
    }
}
```

Issues and possible improvement

Maybe created many unnecessary variables, some variables can be reused and some can be removed.

There are some duplicate things in code, some of them can be removed.

Do not have sample correct inputs and outputs, I do not really know the correctness of my code.

Manual

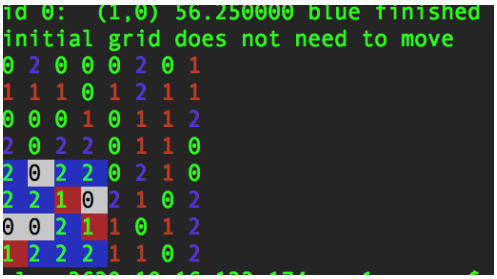
make or mpicc -o assignment assignment.c

mpirun -np {the number of processors} assignment {size of gird} {size of tile} {terminating threshold} {maximum number of iterations}
e.g.mpirun -np 3 assignment 8 2 70 20
using 3 processors, the grid size is 8 by 8, tile size if 2 by 2, coloured cells tile need to reach 70% to terminate, maximum number of iterations are 20.

For only one processor(sequential), the output should contain the processor id and coordinate(row, column) of tile who reach the threshold, the percentage and coloured cell(red or blue), the number of iterations after terminating or print out “initial grid does

not need move” if the initial grid reached the requirement, the entire grid and tile(s) who reach the threshold should be easily seen.

If the initial grid does not need move, it should print out “initial grid does not need to move”



If there are more than one processors, the coordinate(row, column) of tile who reach the threshold is from processor’s perspective(not the coordinate from entire grid’s perspective). It should print out the number of rows of tiles for every processors. The entire grid for parallel computing should also be printed out. In the end it should print out the result of comparing sequential computing with parallel computing(“campared with sequential result, all good!” or specific row and column number and different value).