CMPE 300 – Analysis of Algorithms
Fall 2015

Student:
Behlülcan Mert Çotuk

Instructor:
Tunga Güngör

Programming Project:
Parallel Maze Solver
10/01/2016

1. Introduction

1.a. Terminology:

Maze: A 2 dimensional *nxn* array of integers 0 and 1. At a given position in maze, 0 means there is a wall, whereas 1 means there is a path. A maze is assumed to have two entrance points. Solution is a path from one entrance point to the other one. Therefore, a maze is assumed to have a unique solution.

Cell: A position in the maze.

Wall: A cell that is not a part of the solution.

Path: A cell that is a part of the solution.

Dead-end: A cell that is surrounded by 3 walls. It is impossible to move from a dead-end to an unvisited cell. A dead-end is not a part of the solution.

Solution: A collection of adjacent path cells connecting two entrance points.

Rank: A unique integer in the interval *[0, number of processors)* assigned to each processor by the MPI.

Master Processor: The processor with the rank equal to 0.

Slave Processor: A processor with a rank in the interval *[1, number of processors)*.

p: Number of processors – 1, number of slave processors.

Partition: A *(n/p)xn* cells of the maze.

Neighbour: The lowest row of the upper adjacent processor's partition or the highest row of the lower adjacent processor's partition.

Boundary: A row of a slave processor's partition which is a neighbour for the adjacent slave processor.

1.b. Problem Definition & Overview of the Solution

The problem considered in this project is solving a maze by implementing a parallel algorithm. The sequential algorithm solving the maze iteratively finds and fills all the dead-ends until there are no dead-ends remaining. After that all the adjacent path cells gives us the solution of the maze. In the parallel version of the algorithm, master processor horizontally divides (by rows) the maze into partitions and sends to slave processors. Each slave processor receives the partition sent by the master processor, communicates with adjacent processors to update neighbours, fills the dead-ends considering the neighbour's status, and sends the current partition to master processor. The master processor receives these partitions, updates the maze, and checks whether there are remaining dead-ends. If yes, the same operations are performed by all the processors in the next iteration. If no, the maze is solved.

The nature of the problem is strongly related to graph theory, since mazes containing no loops are also trees[1]. This algorithm is known as a dead-end filling algorithm[2]. Shortest path algorithms also help us to solve a maze. There are also other maze solving algorithms such as random mouse, wall follower, Pledge, Trémaux's algorithm[3]. These algorithms, however, use travelers who cannot see the whole maze at once differing from the dead-end filling and shortest path algorithms[4].

2. Program Interface

In order to run the program, user must install the MPI and be able to compile and run parallel C code in her environment. The code is written and tested in Ubuntu 14.04 LTS. In Ubuntu, user should enter the following command to compile the code:

mpicc -g maze_solver_paral.c -o maze_solver_paral

After compiling, user can run the code with the following command:

mpiexec -n 5 ./maze_solver_paral in1.txt out1.txt

It is assumed that the code is placed under the same directory with input files. Input & output files are passed as arguments where *in1.txt* is the input file and *out1.txt* is the output file to be created. Number of processors is 5 in the above command (1 master processor and p=4 slave processors). In Ubuntu, the user can terminate the program by using Ctrl + C during execution in the console.

3. Program Execution

The input to this program must be a text file containing only integers. The first line of the input file is the dimension *n* of the square maze. It is followed by *n* rows each containing *n* cells (i.e. 0's or 1's). These lines contain all the data for the maze grid. The code reads the input maze into 2D array of integers and applies the dead-end filling algorithm in parallel to solve it. Then, it writes the solution into an output file. The execution has no menu feature or any GUI, but is simple as explained in the previous section. The output file is also a text file containing only integers. The output file contains all the *n* rows each containing *n* cells (i.e. 0's or 1's) of the solved maze array.

4. Input and Output

The first line of the input file is the dimension *n* of the square maze. It is followed by *n* rows each containing *n* cells (i.e. 0's or 1's). These lines contain all the data for the maze grid. The following is an example input for the maze solver:

20

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 1 0 1 1 1 1 1 1 0 1 0 1 1 1 1 1 1 1 0

0 1 0 1 0 1 0 0 0 0 1 0 1 0 0 0 0 0 1 0

[1] https://en.wikipedia.org/wiki/Maze_solving_algorithm
[2] https://en.wikipedia.org/wiki/Maze_solving_algorithm
[3] https://en.wikipedia.org/wiki/Maze_solving_algorithm
[4] https://en.wikipedia.org/wiki/Maze_solving_algorithm

0 1 0 1 0 1 1 1 0 1 1 0 1 1 1 0 1 0 1 0

0 1 0 0 0 0 0 1 0 0 1 0 0 0 1 0 1 0 1 0

0 1 1 1 1 1 0 1 1 1 1 0 1 1 1 1 1 0 1 0

0 0 0 1 0 1 0 1 0 0 1 0 1 0 0 0 0 0 1 0

0 1 1 1 0 1 1 1 0 1 1 1 1 0 1 1 1 0 1 0

1 1 0 1 0 0 0 0 0 1 0 0 0 0 1 0 1 0 1 0

0 0 0 1 1 1 1 1 0 1 1 1 1 0 1 0 1 1 1 0

0 1 0 0 0 0 0 1 0 0 0 0 1 0 1 0 0 0 0 0

0 1 1 1 1 1 0 1 0 1 1 1 1 0 1 1 1 1 1 0

0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0

0 1 1 1 0 1 0 1 1 1 1 1 1 0 1 1 1 0 1 0

0 1 0 0 0 1 0 0 0 0 1 0 1 0 1 0 1 0 1 0

0 1 0 0 0 1 0 0 0 0 1 0 1 0 1 0 1 0 1 0

0 1 0 1 1 1 0 1 1 1 1 0 1 1 1 0 1 1 1 0

0 1 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0

0 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 0

0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0

The output file contains all the *n* rows each containing *n* cells (i.e. 0's or 1's) of the solved maze array. The following is an example output for the maze solver according to the given input above:

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0

0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0

0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 1 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0

0 0 0 1 1 1 0 1 1 1 1 0 1 1 1 0 0 0 1 0

0 0 0 1 0 1 0 1 0 0 1 0 1 0 0 0 0 0 1 0

0 1 1 1 0 1 1 1 0 0 1 1 1 0 1 1 1 0 1 0

1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 1 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0

0 0 0 0 0 0 0 0 0 1 1 1 0 1 1 1 0 1 0

0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 1 0 1 0

0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 1 0 1 0

0 0 0 0 0 0 0 1 1 1 1 0 1 1 1 0 1 1 1 0

0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0

5. **Program Structure**

The pseudocode given in the project description is implemented in parallel. Slave processors get only the partition of the input array. All the operations including reading input and writing output are handled by the master processor. Therefore, all the variables used by a specific processor is declared under the if condition for that processor. For instance, *if(rank == 0) { ... }* declares the variables and executes the statements required by only the master processor. The same philosophy also applies for the slave processors (*else { ... }*).

5.a. Variables

- **#define send_data_tag 2001**

The data tag for the message data sent from the master processor to slave processors.

- **#define return_data_tag 2002**

The data tag for the message data returning from slave processors to the master processor.

- **int rank, num_procs;**

Declared in the main function. Variable *rank* is used to differentiate between different processors. It is initialized by function *MPI_Comm_rank*. Variable *num_procs* is the total number of processes used for the purposes such as distributing the data among the processors. It is initialized by function *MPI_Comm_size*.

- **int proc_id;**

Declared under the code fragment for the master processor. It is used as an index for different processors. It is used when sending/receiving data by the master processor.

- **int A[size][size]**;

Declared under the code fragment for the master processor. It represents the maze as a 2D integer array. It is updated after each iteration completed by all slave processors. After the execution is completed successfully, it contains all the output data.

- **int part_array[rows][cols];**

Declared under the code fragment for slave processors. It represents the portion of the maze for each processor that it is responsible for solving.

- **int distribution[num_procs-1];**

Declared under the code fragment for the master processor. It holds the number of rows to send as a partition to each slave processor.

- **int start_rows[num_procs-1];**

Declared under the code fragment for the master processor. It holds the starting position of the partition to be sent in the maze array to each slave processor.

- **int next_itr;**

Declared for both the master processor and the slave processors. It is an integer used as a boolean in the while loop's condition.

5.b. Functions

The pseudocode given in the project description is implemented in parallel by the functions described below.

- **int wall_neighbours(int rows, int cols, int A[][cols], int i, int j)**

This function counts and returns the number of wall neighbour cells of a given cell in a given array. It is called with the maze array as a parameter by the master processor.

- **int wall_neighbours_boundary(int rows, int cols, int A[][cols], int i, int j, int proc_id, int num_procs, int upper_neighbour[cols], int lower_neighbour[cols])**

This function also counts and returns the number of wall neighbour cells of a given cell in a given array. However, it is called with partition arrays, processor id (rank), upper neighbour, and lower neighbour rows as parameters by the slave processors.

- **bool dead_ends_present(int rows, int cols, int A[][cols])**

This function looks for the all cells in a given array iteratively. It returns true if it finds a dead-end. Else it returns false. It is called by only the master processor. It is called once before the while loop just to initialize the boolean for the next iteration. All other calls are done inside the while loop of the master processor. All calls to the function have the maze array as the parameter.

- **int main(int argc, char **argv)**

The main function is divided into two conditions: the master processor and the slave processors.

The master processor first reads the input file into maze array A. Then it horizontally partitions the maze array into equal chunks for each slave processor. It initializes the boolean for the next iteration according to the maze array A. It sends this boolean to slave processors. Then while there exists the next iteration, it sends the partitions to slave processors, receives the incomplete partial solutions, updates the boolean for the next iteration according to the maze array A , and sends it to slave processors. After this while loop ends, it writes the solution to the output file. Both the input file and

the output file are passed as arguments to the main function.

Each slave processor first receives the boolean for the next iteration to start the while loop. Then, it receives the partition it is responsible for solving. It communicates with its lower adjacent processor and upper adjacent processor to fill the relevant neighbour arrays if it has any. After that, it fills the dead ends in the part_array regarding the neighbour status. It sends the incomplete solution for the partition to the master processor. It receives the boolean value for the next iteration from the master processor. If the next iteration value is 1 the same iteration is performed again. If the value is 0, the solution is completed.

6. **Examples**

After the code is compiled it is ready to run as specified above. The following are the example executions of the code for different inputs:

Example 1:
mpiexec -n 5 ./maze_solver_paral in1.txt out1.txt

in1.txt:
20

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 1 0 1 1 1 1 1 1 0 1 0 1 1 1 1 1 1 1 0

0 1 0 1 0 1 0 0 0 0 1 0 1 0 0 0 0 0 1 0

0 1 0 1 0 1 1 1 0 1 1 0 1 1 1 0 1 0 1 0

0 1 0 0 0 0 0 1 0 0 1 0 0 0 1 0 1 0 1 0

0 1 1 1 1 1 0 1 1 1 1 0 1 1 1 1 1 0 1 0

0 0 0 1 0 1 0 1 0 0 1 0 1 0 0 0 0 0 1 0

0 1 1 1 0 1 1 1 0 1 1 1 1 0 1 1 1 0 1 0

1 1 0 1 0 0 0 0 0 1 0 0 0 0 1 0 1 0 1 0

0 0 0 1 1 1 1 1 1 0 1 1 1 1 0 1 0 1 1 1 0

0 1 0 0 0 0 0 1 0 0 0 0 1 0 1 0 0 0 0 0

0 1 1 1 1 1 0 1 0 1 1 1 1 0 1 1 1 1 1 0

0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0

0 1 1 1 0 1 0 1 1 1 1 1 1 0 1 1 1 0 1 0

0 1 0 0 0 1 0 0 0 0 1 0 1 0 1 0 1 0 1 0

0 1 0 0 0 1 0 0 0 0 1 0 1 0 1 0 1 0 1 0

0 1 0 1 1 1 0 1 1 1 1 0 1 1 1 0 1 1 1 0

0 1 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0

0 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 0

0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0

out1.txt:

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0

0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0

0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 1 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0

0 0 0 1 1 1 0 1 1 1 1 0 1 1 1 0 0 0 1 0

0 0 0 1 0 1 0 1 0 0 1 0 1 0 0 0 0 0 1 0

0 1 1 1 0 1 1 1 0 0 1 1 1 0 1 1 1 0 1 0

1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 1 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0

0 0 0 0 0 0 0 0 0 0 1 1 1 0 1 1 1 0 1 0

0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 1 0 1 0

0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 1 0 1 0

0 0 0 0 0 0 0 1 1 1 1 0 1 1 1 0 1 1 1 0

0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0

Example2:

mpiexec -n 5 ./maze_solver_paral in2.txt out2.txt

in2.txt:
45

0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 1 0 1 1 1 1 1 1 0 1 1 1 1 0 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0

0 1 0 1 0 1 0 0 0 1 0 0 0 1 0 1 0 1 0 1 0 0 0 1 0 1 0 0 0 1 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0

0 1 0 1 0 1 1 1 0 1 0 1 1 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 1 1 1 1 0 1 1 1 1 1 1 0

0 1 0 0 0 0 0 1 0 1 0 1 0 0 0 0 0 1 0 0 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 0 0 0 0 1 0 1 0 0 0 0 0

0 1 1 1 1 1 0 1 0 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 0 1 0 1 0 1 0 1 0 1 1 1 1 1 0 1 0 1 0 1 1 1 0

0 0 0 0 0 1 0 1 0 1 0 1 0 1 0 1 0 0 0 1 0 0 0 1 0 1 0 1 0 1 0 0 0 0 0 1 0 1 0 1 0 1 0 1 0 1 0

0 1 1 1 0 1 1 1 0 1 0 1 0 1 0 1 1 1 0 1 0 1 1 1 0 1 1 1 0 1 0 1 1 1 0 1 0 1 0 1 1 1 0 1 0

0 1 0 1 0 0 0 0 0 1 0 0 0 1 0 0 0 1 0 1 0 0 0 0 0 0 0 1 0 1 0 1 0 1 0 0 0 1 0 0 0 0 0 0 1 0

0 1 0 1 1 1 1 1 0 1 1 1 0 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 0 1 1 1 0 1 1 1 1 1 1 0 1 0 1 1 1 0

0 1 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0

0 1 1 1 1 1 0 1 1 1 1 1 0 1 1 1 1 1 0 1 0 1 1 1 0 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 0

0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 1 0 1 0 0 0 1 0 1 0 1 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0

0 1 1 1 0 1 0 1 0 1 1 1 1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1 1 1 1 0 1 1 1 0

0 1 0 0 0 1 0 1 0 1 0 1 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 1 0 1 0 0 0

0 1 0 1 1 1 0 1 1 1 0 1 0 1 0 1 1 1 1 1 0 1 1 1 0 1 1 1 1 1 1 1 1 0 1 1 1 0 1 1 1 0 1 1 1 0

0 1 0 1 0 0 0 1 0 0 0 1 0 1 0 0 0 0 1 0 0 0 1 0 1 0 0 0 0 0 1 0 1 0 0 0 1 0 1 0 0 0 1 0 1 0

0 1 1 1 0 1 1 1 0 1 1 1 1 1 0 1 0 1 1 1 0 1 0 1 1 1 1 1 0 1 0 1 0 1 0 1 1 1 0 1 0

0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 1 0 1 0 0 0 1 0 0 0 0 0 0 0 1 0 1 0 1 0 1 0 0 0 1 0

0 1 1 1 0 1 0 1 1 1 0 1 0 1 1 1 0 1 1 1 0 1 1 1 1 1 0 1 1 1 1 1 0 1 0 1 1 1 0 1 0 1 1 1 0

0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 1 0 1 0 1 0 0 0 0 0 1 0 1 0

0 1 0 1 1 1 1 1 0 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 0 1 1 1 0 1 0 1 1 1 0 1 0 1 1 1 1 1 0 1 0

0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 1 0 1 0 0 0 0 0 1 0

0 1 1 1 1 1 0 1 1 1 0 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 0 1 0 1 1 1 1 1 0 1 0

0 1 0 1 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 0 0 1 0 1 0

0 1 0 1 0 1 1 1 0 1 0 1 0 1 1 1 1 1 0 1 1 1 1 1 1 1 0 1 1 1 1 1 0 1 0 1 0 1 1 1 0 1 0 1 0

0 1 0 0 0 1 0 0 0 1 0 1 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 1 0 0 0 1 0 1 0 1 0 0 0 1 0 1 0 1 0

0 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 0 1 1 1 1 1 0 1 0 1 1 1 1 1 0 1 1 1 0 1 0 1 0 1 0

0 1 0 1 0 0 0 1 0 1 0 1 0 0 0 1 0 1 0 1 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 1 0

0 1 1 1 0 1 1 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1 0 1 1 1 1 1 0 1 1 1 1 1 1 1 1 0 1 0 1 1 1 0 1 0

0 1 0 0 0 1 0 0 0 1 0 1 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 0 0 1 0 1 0 1 0 0 0 1 0

0 1 0 1 1 1 1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1 1 1 1 0 1 1 1 1 1 0 1 1 1 0 1 1 1 0

0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 1 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0

0 1 1 1 1 1 0 1 1 1 1 1 1 1 0 1 0 1 0 1 1 1 0 1 0 1 1 1 0 1 1 1 0 1 1 1 1 1 1 1 1 0 1 1 1 0

0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 1 0 1 0 0 0 1 0 1 0 0 0 1 0 1 0 0 0 1 0 0 0 0 0 1 0 0 0 1 0

0 1 1 1 0 1 1 1 0 1 1 1 1 1 1 0 1 0 1 1 1 0 1 1 1 0 1 0 1 0 1 0 1 1 1 0 1 1 1 0 1 0 1 1 1 0

0 1 0 0 0 0 0 1 0 1 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 1 0 1 0 1 0 1 0 0 0 1 0 1 0 1 0 1 0 0 0

0 1 1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 0 1 1 1 1 1 1 1 1 0 1 0 1 0 1 0 1 1 1 0 1 0 1 0 1 0 1 0

0 0 0 1 0 1 0 0 0 0 0 1 0 1 0 0 0 1 0 1 0 0 0 1 0 1 0 1 0 1 0 1 0 1 0 0 0 0 0 1 0 1 0 1 0

0 1 1 1 0 1 0 1 1 1 0 1 0 1 0 1 1 1 0 1 0 1 1 1 0 1 0 1 0 1 0 1 1 1 0 1 1 1 0 1 0 1 0 1 0

0 1 0 0 0 1 0 1 0 1 0 1 0 1 0 0 0 1 0 1 0 1 0 0 0 0 0 1 0 1 0 0 0 1 0 1 0 1 0 1 0 1 0 1 0

0 1 0 1 1 1 0 1 0 1 0 1 0 1 1 1 0 1 1 1 0 1 0 1 1 1 1 1 0 1 1 1 1 1 0 1 0 1 1 1 0 1 0 1 0

0 1 0 1 0 0 0 1 0 1 0 0 0 0 0 1 0 1 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 1 0

0 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 0 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

out2.txt:

0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 1 1 1 1 1 1 1 1 1 1 0

0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 1 1 1 1 0 1 1 1 1 0

0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0

0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 1 0 1 1 1 0

0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 1 0 1 0 1 0

0 1 1 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 1 0 0 0 1 0 1 1 1 0 1 0

0 1 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 0 0 1 0 0 0 0 0 1 0

0 1 0 1 1 1 1 1 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 1 1 1 1 1 0 0 0 1 1 1 0

0 1 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0

0 1 1 1 1 1 0 1 1 1 1 1 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0

0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0

0 0 0 0 0 1 0 0 0 0 0 1 1 1 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 1 1 1 0

0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 1 0 0 0

0 0 0 1 1 1 0 0 0 0 0 1 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 1 1 1 0 1 1 1 0

0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 1 0

0 1 1 1 0 0 0 0 0 1 1 1 0 1 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 1 0

0 1 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 1 0

0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 1 1 1 0 0 0 0 0 0 0 1 1 1 1 1 0 1 0 1 1 1 0 0 0 1 1 1 0

0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 1 0 1 0 0 0 0 0 1 0 0 0

0 0 0 1 1 1 1 1 0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 1 1 1 0 1 0 1 1 1 1 1 0 0 0

0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0

0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 1 0 1 1 1 1 1 0 0 0

0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0

0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0

0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0

0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 1 1 1 0 0 0 1 0 0 0

0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0

0 1 1 1 0 0 0 0 0 0 0 0 0 1 1 1 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 1 0 0 0

0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0

0 1 0 0 0 0 0 0 0 0 0 0 1 1 1 0 1 1 1 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0

0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 1 1 1 1 1 0 0 0 0 0 0 0 0 1 0 0 0 1 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 1 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 1 1 1 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 1 0 1 1 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 1 0 1 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 1 1 1 0 1 0 1 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 1 0 0 0 1 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 1 1 1 1 1 0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

7. **Improvements and Extensions**

The code works just as it is expected. The possible future extension can be implementing the bonus part specified in the project description. The weak point of the code may be the parameter names of some function definitions. The strong point of the code is its efficiency. Almost everything is declared and initialized only when and where it is necessary.

The only difference between the expected output and the program output might cause from the number of whitespaces. If you use *"diff"* command when comparing outputs, it is recommended to use *"diff -b"* to ignore whitespaces.

8. **Difficulties Encountered**

The biggest difficulty I encountered was to synchronize the iterations of the master processor and the slave processors. I spent almost the half of my implementaion effort to detect the reason of the deadlocks. After I finally detect that, everything was easier for me.

Another difficulty was to pass 2D arrays as function arguments and returning them in C. This was because of I have not used C for such programming purposes before. However, none of the difficulties changed the plan and had no effect on the final code. Everything works as expected.

9. **Conclusion**

It was really informative to write a parallel C code for a complex task such as solving a maze. It solidified the great amount of the knowledge I obtained in lectures. The code divides the input according to the number of slave processors. Each slave processor works parallel and sequentially solves its partition. The whole array is not input to the slave processors. Therefore, the parallel code is more efficient than the sequential one.

References

1) https://en.wikipedia.org/wiki/Maze_solving_algorithm
2) https://en.wikipedia.org/wiki/Maze_solving_algorithm
3) https://en.wikipedia.org/wiki/Maze_solving_algorithm
4) https://en.wikipedia.org/wiki/Maze_solving_algorithm

Appendices

maze_solver_paral.c

```
/*
Student Name: Behlülcan Mert ÇOTUK
Student Number: 2011400294
Compile Status: Compiling
Program Status: Working
Notes: There might be differences in whitespaces between the expected output and the program's output.
If you use "diff" command when comparing outputs, it is recommended to use "diff -b" to ignore whitespaces.
*/

#include <stdio.h>
#include <stdbool.h>
#include "mpi.h"
#include <stdlib.h>

#define send_data_tag 2001
#define return_data_tag 2002


// counts the number of wall neighbours of a cell
int wall_neighbours(int rows, int cols, int A[][cols], int i, int j) {

    // number of walls around the cell
    int walls_around = 0;

    // check up
    if((i-1)>=0) {
        if(A[i-1][j]==0) walls_around++;
    }

    // check down
    if((i+1)<rows) {
        if(A[i+1][j]==0) walls_around++;
    }

    // check left
    if((j-1)>=0) {
        if(A[i][j-1]==0) walls_around++;
    }

    // check right
    if((j+1)<cols) {
        if(A[i][j+1]==0) walls_around++;
    }

    return walls_around;
```

```
}

// counts the number of wall neighbours of a cell also for boundary conditions
int wall_neighbours_boundary(int rows, int cols, int A[][cols], int i, int j
                , int proc_id, int num_procs, int upper_neighbour[cols], int lower_neighbour[cols])
{

   // number of walls around the cell
   int walls_around = 0;

   // check upwards
   if((i-1)>=0) {
      if(A[i-1][j]==0) walls_around++;
   } else {
      if(proc_id > 1) {
         if(upper_neighbour[j]==0) walls_around++;
      }
   }

   // check downwards
   if((i+1)<rows) {
      if(A[i+1][j]==0) walls_around++;
   } else {
      if(proc_id < num_procs-1) {
         if(lower_neighbour[j]==0) walls_around++;
      }
   }

   // check left
   if((j-1)>=0) {
      if(A[i][j-1]==0) walls_around++;
   }

   // check right
   if((j+1)<cols) {
      if(A[i][j+1]==0) walls_around++;
   }

   return walls_around;
}

// checks whether there are remaining dead ends
bool dead_ends_present(int rows, int cols, int A[][cols]) {

   int i, j;
   for( i = 0; i < rows; i++) {
      for( j = 0; j < cols; j++ ) {
         if(A[i][j]==1 && wall_neighbours(rows,cols,A,i,j)==3) return true;
      }
   }
   return false;
}
```

```
int main(int argc, char **argv)
{
   // initializations
   int rank, num_procs;
   int proc_id;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
   MPI_Status status;


   // master proc
   if(rank == 0) {

      // read size of the input from file
      FILE *input = fopen(argv[1], "r");
      size_t size;
      fscanf(input, "%zd", &size);

      // create the main array A
      size_t i, j;
      int A[size][size];

      // fill A from file
      for( i = 0; i < size; ++i ) {
         for( j = 0; j < size; ++j ) {
            fscanf( input, "%d", &A[i][j] );
         }
      }
      fclose(input);

      // partition array rows almost equally
      int distribution[num_procs-1];
      int start_rows[num_procs-1];
      int current_start_row = 0;
      int quotient = size / (num_procs-1);
      int remains = size % (num_procs-1);
      for(proc_id = 1; proc_id < num_procs; proc_id++) {
         start_rows[proc_id] = current_start_row;
         distribution[proc_id] = quotient;
         if(remains != 0) {
            distribution[proc_id]++;
            remains--;
         }
         current_start_row += distribution[proc_id];
      }


      // send next iteration to slave procs
      // next_itr is used as a boolean (i.e. 1 means true, 0 means false) to synchronize the loops of
```

```
master and slave procs
        int next_itr;
        if(dead_ends_present(size,size,A)) next_itr = 1;
        else next_itr = 0;
        for(proc_id = 1; proc_id < num_procs; proc_id++) {
           MPI_Send( &next_itr, 1, MPI_INT,
                        proc_id, send_data_tag, MPI_COMM_WORLD);
        }

        while(next_itr) {

           // send partitions to slave procs
           for(proc_id = 1; proc_id < num_procs; proc_id++) {
              MPI_Send( &distribution[proc_id], 1, MPI_INT,
                        proc_id, send_data_tag, MPI_COMM_WORLD);
              MPI_Send( &size, 1, MPI_INT,
                        proc_id, send_data_tag, MPI_COMM_WORLD);
              MPI_Send( &A[start_rows[proc_id]][0], distribution[proc_id]*size, MPI_INT,
                        proc_id, send_data_tag, MPI_COMM_WORLD);
           }


           // recieve partitions and update A
           for(proc_id = 1; proc_id < num_procs; proc_id++) {
              MPI_Recv( &A[start_rows[proc_id]][0], distribution[proc_id]*size, MPI_INT,
                 proc_id, return_data_tag, MPI_COMM_WORLD, &status);
           }

           // send next iteration to slave procs
           // next_itr is used as a boolean (i.e. 1 means true, 0 means false) to synchronize the loops
of master and slave procs
           if(dead_ends_present(size,size,A)) next_itr = 1;
           else next_itr = 0;
           for(proc_id = 1; proc_id < num_procs; proc_id++) {
              MPI_Send( &next_itr, 1, MPI_INT,
                           proc_id, send_data_tag, MPI_COMM_WORLD);
           }
        }

        // write solved grid to output file
        FILE *output = fopen(argv[2], "w");
        for(i=0; i<size; i++) {
           for(j=0; j<size; j++) {
              fprintf(output, " %d", A[i][j]);
           }
           fprintf(output, "\n");
        }
        fclose(output);
     }
```

```
// slave procs
else {

   int next_itr;
   MPI_Recv( &next_itr, 1, MPI_INT,
         0, send_data_tag, MPI_COMM_WORLD, &status);


   while(next_itr) {

      // recieve partition from master proc
      int rows, cols;
      MPI_Recv( &rows, 1, MPI_INT,
            0, send_data_tag, MPI_COMM_WORLD, &status);
      MPI_Recv( &cols, 1, MPI_INT,
            0, send_data_tag, MPI_COMM_WORLD, &status);
      int part_array[rows][cols];
      MPI_Recv( &part_array[0][0], rows*cols, MPI_INT,
               0, send_data_tag, MPI_COMM_WORLD, &status);

      // create neighbour cells array for boundary conditions
      size_t i, j;
      int upper_neighbour[cols], lower_neighbour[cols];
      for(j = 0; j < cols; j++) {
         upper_neighbour[j] = 0;
         lower_neighbour[j] = 0;
      }

      // communicate slave procs to update neigbour cells array
      if(rank % 2 == 1) {
         if(rank + 1 < num_procs) {
            MPI_Send( &part_array[rows-1][0], j, MPI_INT,
               rank + 1, send_data_tag, MPI_COMM_WORLD);
            MPI_Recv( &lower_neighbour[0], j, MPI_INT,
               rank + 1, send_data_tag, MPI_COMM_WORLD, &status);
         }
         if(rank > 1) {

            MPI_Recv( &upper_neighbour[0], j, MPI_INT,
               rank - 1, send_data_tag, MPI_COMM_WORLD, &status);
            MPI_Send( &part_array[0][0], j, MPI_INT,
               rank - 1, send_data_tag, MPI_COMM_WORLD);
         }
      } else {
         if(rank > 1) {
            MPI_Recv( &upper_neighbour[0], j, MPI_INT,
               rank - 1, send_data_tag, MPI_COMM_WORLD, &status);
            MPI_Send( &part_array[0][0], j, MPI_INT,
               rank - 1, send_data_tag, MPI_COMM_WORLD);
         }
         if(rank + 1 < num_procs) {
```

```
            MPI_Send( &part_array[rows-1][0], j, MPI_INT,
               rank + 1, send_data_tag, MPI_COMM_WORLD);
            MPI_Recv( &lower_neighbour[0], j, MPI_INT,
               rank + 1, send_data_tag, MPI_COMM_WORLD, &status);
         }
      }

      // mark as wall if it is a dead end
      for(i=0; i<rows; i++) {
         for(j=0; j<cols; j++) {
            if(part_array[i][j]==1 && wall_neighbours_boundary(rows,cols,part_array,i,j
            ,rank,num_procs,upper_neighbour,lower_neighbour)==3) {
               part_array[i][j] = 0;
            }
         }
      }

      // send partition to master proc
      MPI_Send( &part_array[0][0], rows*cols, MPI_INT,
            0, return_data_tag, MPI_COMM_WORLD);

      // recieve next iteration from master proc
      // next_itr is used as a boolean (i.e. 1 means true, 0 means false) to synchronize the loops
of master and slave procs
      MPI_Recv( &next_itr, 1, MPI_INT,
         0, send_data_tag, MPI_COMM_WORLD, &status);
   }
}

MPI_Finalize();
return 0;
}
```