Homework 8: Parallel and Distributed Computing

DUE: See exact time on blackboard

ICSI 520

Problem: Game of Life simulation

In this homework you will implement a solution to the Game of Life Problem in MPI, and you will test your solution for different problem sizes and different numbers of nodes.

The Game of Life is a simple example of a class of problems that can be modeled using cellular automata; some physical and biological systems that evolve over time can be modeled using cellular automata. The problem space is divided into a number of cells each of which is a finite state machine. Each cell's evolution is computed in steps of time; each cell makes one state change, then each cell makes the next state change, and so on.

In the Game of Life, each cell, represented by an (i,j) element in an MxN matrix, contains an organism that is either dead or alive (represented by a 0 or 1). At each step in the game, a cell's state changes based on its current state and the state of its neighbors. The following are rules for a cell's state transition at each step:

1. A live cell with zero or one live neighbors dies from loneliness.
2. A live cell with four or more live neighbors dies due to overpopulation.
3. A dead cell with two or three live neighbors becomes alive.
4. Otherwise, a cell's state stays unchanged.

Using MPI, write a message passing version of the game of life that takes three command line arguments: two dimension values (M and N) for the matrix; and K the number of iterations as input. The easiest way to parallize this program is to have one process for each cell in the MxN matrix. You are welcome to try another way of dividing up the matrix if you'd like (either way is fine). No matter how you decide to distribute the matrix, you should choose one process to be the master (0 is a good choice), and the others to be the slaves. The master should participate as a cell in the MxN matrix (or as a set of cells), but is also responsible for sending slaves their original values and receiving the final result from the slaves and printing it to stdout.

You should also have a debug mode where each slave sends the master its value at the end of a round, and the master after receiving values from all slaves, prints out the MxN matrix of round i values. This will help you debug your solution.

When your program is **not** running in debug mode, slaves should not send the master their intermediate results.

At each step, each cell needs to exchange information with its neighbors. Keep in mind that a cell in the middle has eight neighbors, and a cell in a corner has only three neighbors, and a cell on an edge has five neighbors. Make sure that at each iteration i, each process gets values from each of its neighbors corresponding to their ith state, and that you solution is deadlock free.

Here is some sample output from a run of my program with a 6x4 matrix for two rounds, with debugging turned on (I just use a compile-time flag, #define DEBUG 1, to turn it on/off):

```
Start:
------
0 0 0 1
0 0 0 1
0 1 0 0
1 1 1 0
0 1 0 0
0 1 0 0

Round 0:
-------
0 0 1 0
0 0 1 0
1 1 0 1
1 0 1 0
0 0 0 0
1 0 1 0

Round 1:
-------
0 1 0 1
1 0 1 1
1 0 0 1
1 0 1 1
1 0 1 1
0 1 0 0
```

Once your program works, you should evaluate its performance when run on different numbers of host machines (for example, 2, 4, 8, 16, ...), for different sized matrices (two or three different sized MxN matrices should be fine).

Serial and Distributed Implementations

You do not need a serial implementation for this homework however it is best that you first implement this problem in serial.

Use timing code to measure timings for this code.

Results

Number of cores (p)	**Matrix size**	**Distributed Execution time**
Depends on your implementation	2x2	
	4x2	
	4x4	
	4x8	
	8x8	
	8x2	
	2x9	
	2x10	
	10x4	
	4x10	
	6x4	
	9x2	
	10x6	
	8x16	
	8x10	
	9x7	
	8x3	
	10x9	

1. Homework8_Serial
   a. Include all of the source files for serial implementation
   b. Binaries
   c. Input and output files
2. Homework8_Distributed
   a. Include all of the source files for Distributed implementation
   b. Binaries
   c. Input and output files
3. A document explaining the speedup and the tabular results
   a. Here you will explain for what value of p you are getting the most speedup
4. All of the above should be in one zip/tar following the naming convention noted below.

Name your zip/tar/rar/7z folder as FirstName_LastName_Homework8. You will lose points if you don't follow this convention

Grading

Distributed Implementation	50%
Correct execution Distributed	45%
Code clarity	5%

Do not make up the speedup or the timing tables. You will get a 0 for all homework assignments (including previous).

Code clarity includes

- Use of variable names
- Function naming
- File names
- Formatting
- Comments where needed

Late submissions are subject to at least 20% grade deduction.

Reminder: If one or any of your binaries don't execute on student.rit.albany.edu, you get 0 points for this and any other homework where that happens.