CptS 411, Programming Project 2: Conway's Game of Life

Preamble: You are allowed to work individually or in teams of size two each, for this project. Make only one full submission for each team project (including the Cover Sheet). The other person makes an "empty" submission with just the Cover Sheet (from the website) indicating what your team is and any other appropriate acknowledgements.

Project Description

In this project you will implement the *Conway's Game of Life* in MPI. The game is a cellular automaton containing $m*n$ cells arranged as m rows and n columns.

Cell Neighborhood: Since the automaton is structured as a grid (or mesh), each internal cell will contain exactly 8 neighbors: {N,S,E,W,NE,NW,SE,SW}. For cells that are on the boundary of the automaton (first row/column, last row/column), neighborhood will include wrapped around neighbors from the other end of the matrix – for instance, the first row will treat the last row as its preceding row, while the last column will treat the first column as its preceding column. The corner cells will use the corresponding cell in the diagonally opposite corner as substitute for their corresponding missing diagonal neighbor entries.

Cell States and Generations: Each cell can be in one of the two states at any given point of time – *alive* or *dead*. In the initialization step, we assign the state for each cell by a toss of a coin (i.e., random with equal probability). After that, game simply keeps progressing from one "generation" to another without needing any further user input.

During the k^{th} generation, all cells determine their respective states based on the following simple **rules**:

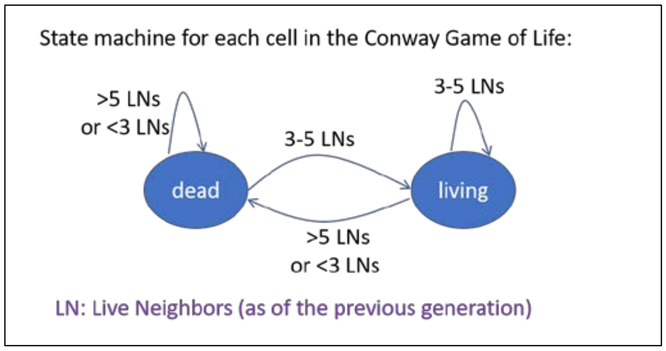
i) A living cell that has less than 3 living neighbors dies (as if caused by underpopulation, or loneliness, or simply boredom!);

ii) A living cell with more than 5 living neighbors also dies (as if caused by overcrowding);

iii) A living cell with between 3 and 5 living neighbors lives on to the next generation;

iv) A dead cell will come (back) to life, if it has between 3 and 5 living neighbors.

Therefore, the decision to live or die obviously depends on the states of a cell's neighbors and its own current state.


State machine for each cell in the Conway Game of Life:
>5 LNs or <3 LNs
3-5 LNs
3-5 LNs
dead
living
>5 LNs or <3 LNs
LN: Live Neighbors (as of the previous generation)

As a rule, we will use the states from *the previous* generation to make the decisions for the current generation. This implies that the states of cells in the current generation can be updated independent of one another and in any arbitrary order without affecting the output.

Your task for this assignment is to write an efficient implementation for the Conway's Game of Life in C/C++/MPI and analyze its scalability (both analytically and empirically). For the purpose of this assignment, we will make the following assumptions:

- that the automaton's matrix is always a **square** with *n* rows and *n* columns;
- that *n>p* and *n* is divisible by *p* (the number of processes);
- that the number of generations to simulate is specified by the user at input. Let this parameter be denoted by *G*;

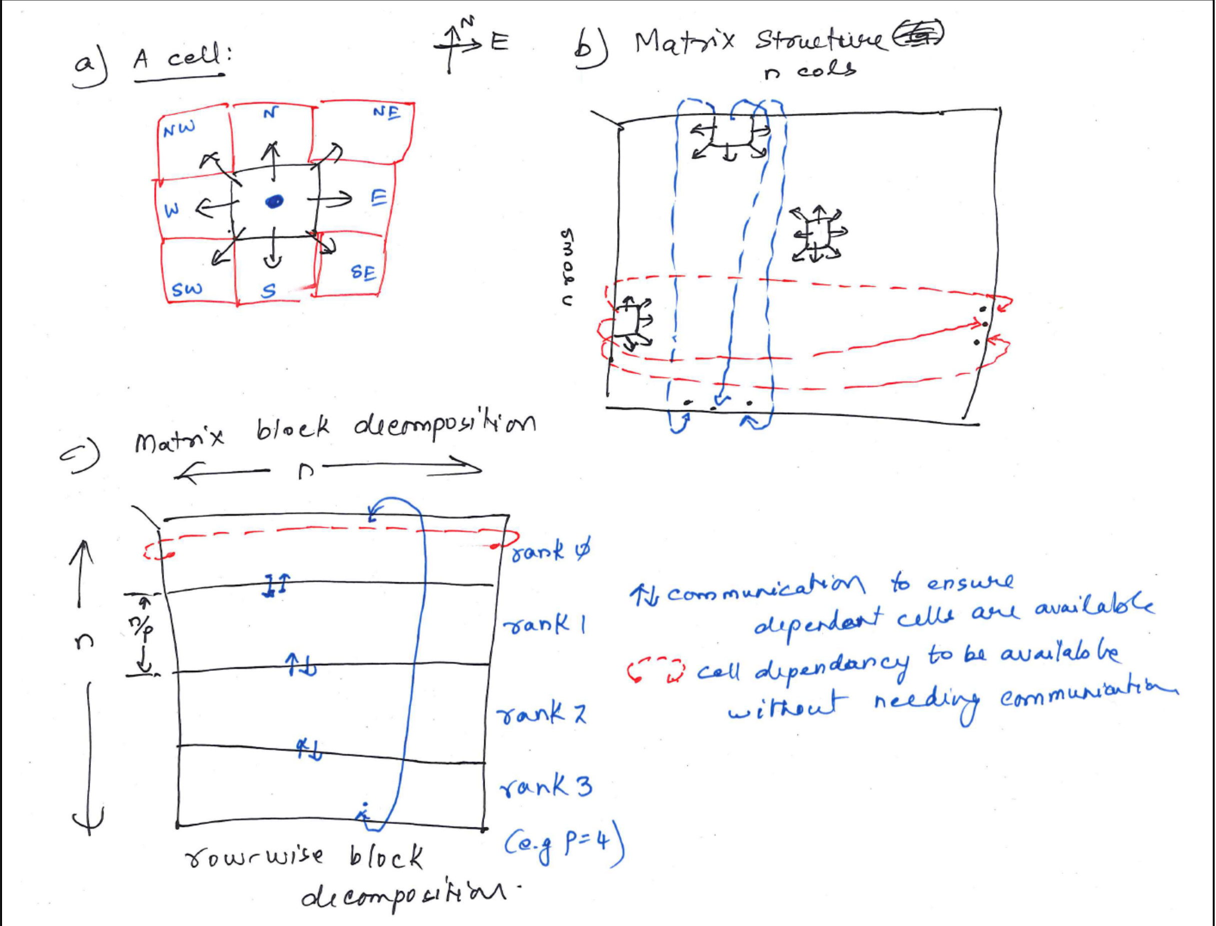

a) A cell:
NW
N
NE
W
E
SW
S
SE
N
E
b) Matrix Structure
n cols
n rows
c) Matrix block decomposition
n
n
n/p
rank 0
rank 1
rank 2
rank 3
(e.g p=4)
rowwise block decomposition.
communication to ensure dependent cells are available
cell dependancy to be available without needing communication

Fig. 1. Illustration of the Game of Life. Part a shows the cell level dependencies, and part b shows the dependencies at the matrix level. Part c shows one recommended approach for parallel implementation.

Parallel implementation:

There are multiple ways in which the Game of Life can be parallelized. One way is to decompose the input matrix into roughly equal sized smaller, nonoverlapping squares (or grids) such that each process owns a distinct, part of the matrix. *Another way is to decompose the matrix into disjoint batches of rows* *(or columns) such that each process gets n/p rows (or columns).* The latter approach, which is typically referred to as *block decomposition*, is better suited for implementation in distributed memory for these kinds of problems (why?). So we will use that approach in the implementation. Note that in both approaches, each rank will get n^2/p cells.

The pseudocode for your algorithm is as follows:

Input: User specifies parameters *n, G*

Initialization:

- ***GenerateInitialGoL()***[1]*:* This function should generate the initial matrix in parallel so that by the end of the call, the entire matrix is generated and stored in a distributed manner (with rank *i* generating and subsequently, also owning rows [i*(n/p) … (i+1)*(n/p)-1]. To do this, there are two steps:
    - Rank 0 first generates *p* random numbers (in the interval of 1 and BIGPRIME[2]) and **distributes** them such that the *i*th random number is handed over to rank *i*.
      (Think of what MPI function you would use to do this distribution.)
    - Next, using the assigned random number as the "seed", each rank (locally) generates a distinct sequence of n^2/p random values, again in the same interval. Each generated random number is used in the following manner to fill the initial matrix:
      if the *k*th random number is even, then the *k*th cell being filled in the local portion of the matrix is marked with status=Alive;
      otherwise its state=Dead.
      (This is one way to generate the matrix randomly. If you prefer to do it other ways, that's fine too but make sure you use different random seeds in different processes to avoid the problem of generating matrix replicas across processes.) Also note that if you initialize this way, the initially filled matrix is not necessarily compliant with the rules of the GoL. That's okay. You will fix it in the first iteration of your simulation. Read on…
- ***Simulate()****:* This function actually runs the Game of Life. The simulation is done for *G* generations (i.e., iterations). Within each iteration, rank *i* determines the new states for all the cell that it owns. To determine the new state of a cell, write a function called *DetermineState()* that takes a cell coordinate and returns its new state (alive or dead) based on the GoL rules described above. For this to happen, however, you need to first make sure all cell states have access to their neighboring entries – i.e., do the necessary communication to satisfy these dependencies *a priori*. You also need to ensure that all processes are executing the same

[1] Note that I have just mentioned the function names here. For consistency, I would like all of you to use the same function names. You are free to decide on the arguments as necessary in your implementation.
[2] Some reasonably big prime number – e.g. 93563, 68111, etc.

generation at any given time. This can be done using a simple *MPI_Barrier* at the start of every generation.

- ***DisplayGoL()****:* As the simulation proceeds it is desirable to display the contents of the entire matrix to visualize the evolution of the cellular automaton. However, doing this after every step of simulation could become cumbersome for large *n*. Therefore, we will do this infrequently – i.e., display once after every *x* number of generations. I will let you decide the value of *x* based on the timings you see for a specific input size. To do the display, write a function called *DisplayGoL()* which first aggregates (i.e., gathers) the entire matrix in rank 0 ("root") and then displays it. (Note, displaying from every separate rank independently is possible but could possibly shuffle up the prints out of order.) As an alternative to each process sending its entire local matrix to the root you could choose to send only the alive entries. This approach would provide some communication savings if the matrix is sparse with very few alive entries. However for simplicity if you want to just send the whole local matrix that is OK for this assignment.

Reporting

- Timing: Your code should measure the following times for reporting purposes:
    - Total runtime (excluding the time for display) to run the simulation for the user-specified *G* generations
    - Average time taken to execute a single generation (excluding time for display)
    - Sum up the time for all communication steps (i.e., if you are using say *MPI_Barrier*, *MPI_Send, MPI_Recv*, *MPI_Gather*, *MPI_Scatter*, *MPI_Bcast*, *MPI_Reduce,* etc.), select the maximum of that time across all processes, and report that as the "total communication time".
    - => "Total computation time" will then be (Total runtime – Total communication time).

  These timings should be reported before the program terminates.

Experiments

To test the scalability of your code, devise the following experiments: Measure the average time per generation (excluding display time) as a function of the number of processes, for varying input sizes but a fixed number of generations (G). Think of building a table for this purpose, with rows for different input sizes (values of n^2) and columns for different number of processes (*p*). To change the input size, grow *n* in powers of 2, starting from an input that is as small as 4x4 and going as large as 2^{10}x2^{10} or more as dictated by your runtime. Vary the number of processes in powers of 2: e.g., {1,2,4,8,…, 64} in our cluster.

Using the above table, generate three plots:

i) **Speedup** in the Average runtime per generation (Y-axis) vs. Number of processes (X-axis) – with different curves for different input sizes (nxn);

ii) **Efficiency** curves for the above chart;

iii) Plot the breakdown (in %) of the total runtime to indicate how much of it was spent in "Computation" (i.e., the time to update the local matrix and anything else that is not included any communication) vs. "Communication" (i.e., the sum of the times spent in communicating primitives). Show this breakdown for varying number of processes for a fixed input size.

Please make sure you follow the templates provided in the lecture notes, to show your runtime/speedup/efficiency results:
https://eecs.wsu.edu/~ananth/CptS411/Lectures/ParallelPerformance_example.pdf

Interpretation: Briefly state your observations about your results – do they meet your analytical expectations? If not, why not? Do you see ways to optimize this further?

Deliverables (zipped into one zip file with your name(s) on it):

i) Cover page mandatory

ii) Full source code inside a source_code folder

iii) Report in PDF (preferred) that shows all scaling results and your interpretation of those results.