Deadline: October 26th, 2021

1. Implement Game of Life using MPI (10 points).

Given *M* and *N*, create a matrix of dimensions *M* and *N* where each cell value is randomly initialized to either a zero or one. You are also given a parameter *T* which is the number of timesteps. At each time step, the value of the cell is updated as follows. A cell is considered alive (alive means cell value is 1) if the cell and at least two of its neighbors were alive in timestep *t-1* OR if the cell was dead, but at least four of its neighbors were alive in timestep *t-1*. The snippet showing the core computation is given below:

```
for (int outer = 0; outer < TIME_STEPS; ++outer)
{
    for (int i = 0; i < N; ++i)
    {
        for (j = 0; j < M; ++j)
        {
            int neigh_count = inp_cells[i - 1][j - 1] + inp_cells[i - 1][j] + inp_cells[i - 1][j + 1] +
                              inp_cells[i + 1][j - 1] + inp_cells[i + 1][j] + inp_cells[i + 1][j + 1] +
                              inp_cells[i    ][j - 1] +                       inp_cells[i    ][j + 1] ;
            if(inp_cells[i][j]==1)
            {
                    out_cells[i][j] = neigh_count>=2? 1 :0;
            }
            else
            {
                    out_cells[i][j] = neigh_count>=4? 1 :0;
            }
        }
    }
swap(out_cells, inp_cells);
}
```

Deliverable: Just MPI code. The submission should be a .c or .cpp file. M, N, and T should be accepted as command line arguments (or using #define at the top of the file). **The submission should be complete, which means that one should be able to compile and run the code.**

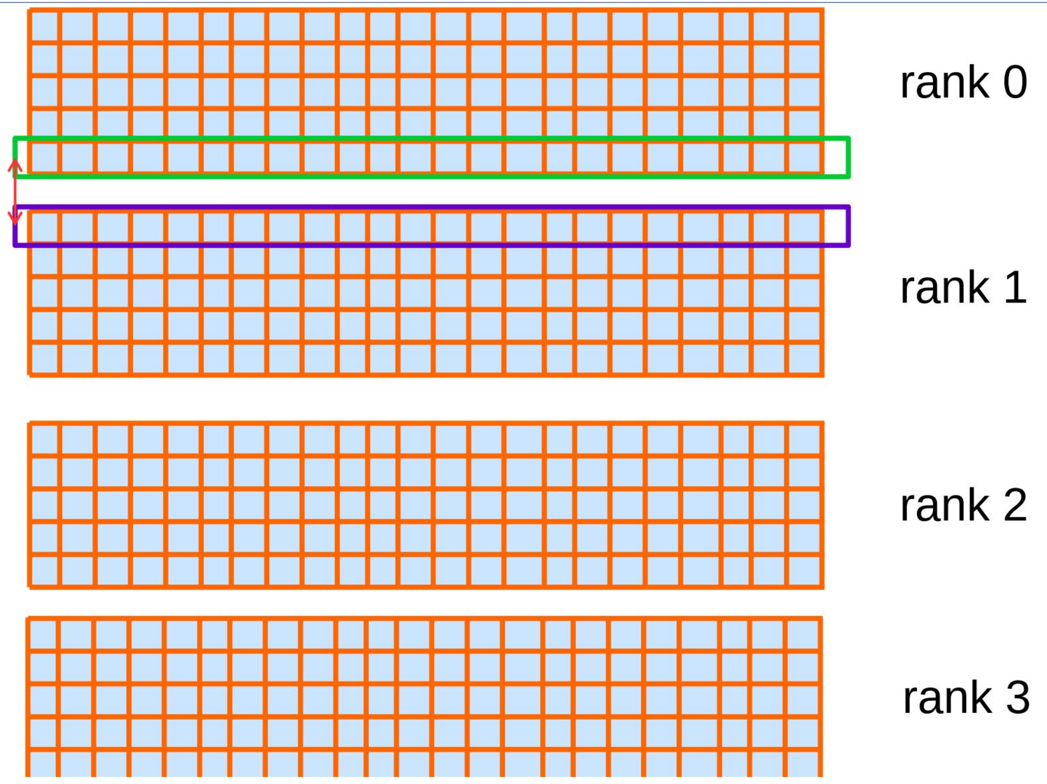
HINTS:

1. **Follow the instruction on CANVAS (Files > Using cluster > MPI > How to run MPI programs_v2.pdf) to run the code. Directly using SRUN on the cluster will not work as expected**
2. **Just as a thinking exercise, imagine that instead of looking at all neighbors, for now, just look at the neighbor below (your code should look at all neighbors, but this exercise is just to help you think). Ignore the contributions from all**

other neighbors (for now). How do we implement this? We can split the rows of the matrix into row panels, each of size (M/P), where P is the number of processors. Each processor owns one row panel.

Let's assume that there are 128 rows and 4 processors. Each processor will own (128/4) = 32 rows. Let's look at it from the perspective of Rank 1. It owns rows from 32 to 63. However, to process row 63, we need row 64 from the previous iteration, which is owned by Rank 2. This means Rank 2 has to send row 64 to rank 1. So how do we tie this all together? Every processor will allocate a buffer of size (33 * N). The 32*N part is for the rows it owns (in our example, rows 32 to 63 for rank 1) and 1*N for holding the data which will be sent by the adjacent processor (in our example, rank 2 will send rank 1 the 64th row). Now the question is when should we exchange the data. We should exchange the data at the beginning of each iteration of the T loop (you can also do it at the end of the T loop, depending on your logic).

Now extend your implementation by considering both top and bottom neighbors. Each processor has to reserve a buffer of size 34*N (32 for the rows it owns, 1 for receiving the top row data which the neighboring top node will send, and 1 for receiving the bottom row data which the neighboring bottom node will send). Once you implement this, you will see that handing the rest of the neighbors is trivial.


rank 0
rank 1
rank 2
rank 3