Cmpe 300: MPI Project
Due: December 20 @17:00

1 Cellular Automata

A cellular automaton is a discrete computational model used in the study of complex systems. It consists of an N-dimensional orthogonal grid called the "map", and rules of evolution. These alone define our complex environment.

We then assign some initial values to the locations on the map, and make our system evolve by our set of rules via simulation:

1. The map's state at time $t = 0$ is as initialized.
2. To calculate the map's state at time $t = 1$, we look at the state at time $t = 0$. For each cell on the map at time $t = 1$, we look at the same cell and its neighbors at time $t = 0$.
3. We keep on simulating like this until time $t = T$.

Now, you may think that this model is not complex at all, and you would be right! In analyzing complex systems, the key is to have a very *simple model* (i.e. a model with a very simple setup and rules of evolution), for an *outcome that has a complex pattern.* This will then allow the researcher to come up with simple explanations to the emergent patterns in a complex system.

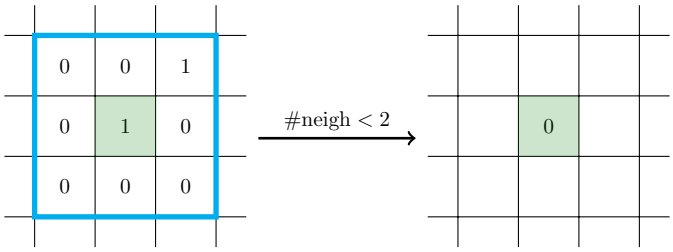
1.1 Game of Life

Our focus will be on the particular cellular automaton called the (Conway's) Game of Life, devised by J. H. Conway in 1970. Just "Game" for short. In the Game, we have a 2-dimensional orthogonal grid as a map (i.e. a matrix). Each cell on the map can either contain a creature (1) or be empty (0).

The 8 cells that are immediately around a cell are considered as its neighbors. Then, the rules of the Game are as follows:

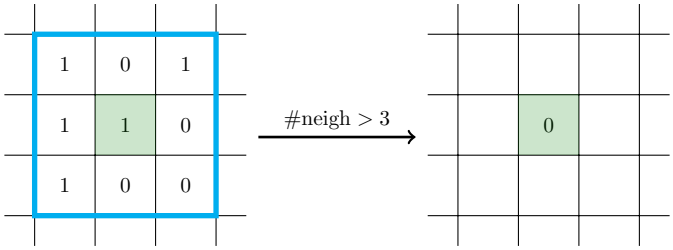
1. Loneliness kills: A creature dies (i.e. the cell becomes empty) if it has less than 2 neighboring creatures.

   The following figure depicts a lonely creature in the green cell with a single creature in its neighborhood (the blue square), not counting the creature itself. It passes away on the next iteration. It would keep on living if it had one more neighboring creature.

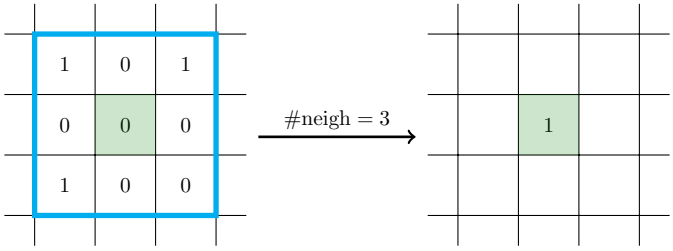


   The neighbors on the right hand side are left empty, since we cannot tell their state without knowing about their neighbors.

2. Overpopulation also kills: A creature dies (and becomes empty) if it has more than 3 neighboring creatures. See the following example. Note that the creature would not die if it had one less neighboring creature.



3. Reproduction: A new life appears on an empty cell if it has exactly 3 neighboring creatures. See the following example. Note that the creature would not be born if the cell had one more or one less neighboring creature.



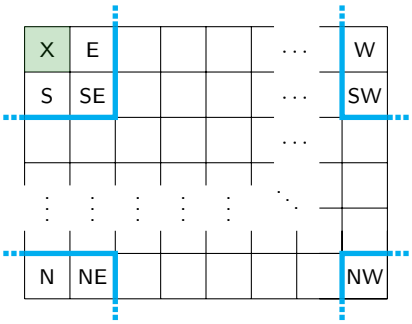
4. In any other condition, the creatures remain alive, and the empty spaces remain empty.

1.2 Boundaries (edges and corners)

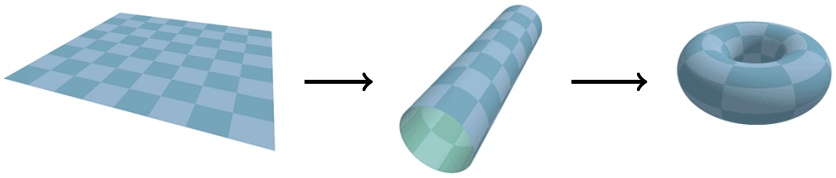
Since the cells at the boundaries do not have all the 8 neighbors, special considerations must be done for them. Generally, in the Game of Life and most of the other cellular automata, there are these two options for the boundaries:

1. **Cutoff:** The missing neighbors are taken as 0, i.e. emptiness in the Game. In this case, the corners can have at most 3 non-empty neighbors, and the edges can have at most 5.

2. **Periodic (20 pts bonus):** The neighbors are not missing! They are just at the opposite end of the map. See the following image for the neighborhood (with the blue "square") of the top left corner cell X. Letters N, S, E, and W denote the neighbors of X at North, South, East, and West. For instance, see how the "northern" neighbor (N) of X is at the bottom of the 2D array.



The map is called *toroidal* in this case. To see how a rectangle bends into a torus, see the animated version of the following images via the link below:



https://plus.maths.org/content/sites/plus.maths.org/files/news/2015/abel/torus_from_rectangle.gif

Note In your project, please pick either the cutoff- or periodic-boundaries, and implement *only* that. In other words, do not implement both of them.

2 Parallel simulation of the Game

In this project, we will simulate the Game using a parallel algorithm. For that we will use an implementation of the Message Passing Interface (MPI) standard called the Open MPI. It will provide us with multiple processes to work with, and functions to send/receive messages among them for the data exchange. Download the latest stable version (4.0.2) via the following link:

https://download.open-mpi.org/release/open-mpi/v4.0/openmpi-4.0.2.tar.gz

Your task is to write an MPI program with 1 manager process and C worker processes. The manager will distribute the simulation task to the workers, and aggregate the results. The workers will, well, work on the task, and occasionally communicate with each other.

2.1 Compiling and running

An MPI source code written in C Language, say `game.c`, can be compiled into the executable `game` with the following:

```
mpicc game.c -o game -lm
```

The `-lm` argument links the math libraries, in case you need `math.h`.
For a C++ source code like `game.cpp`, use:

```
mpic++ game.cpp -o game
```

A compiled MPI program `game`, can be run with the following:

```
mpirun -np [M] --oversubscribe ./game input.txt output.txt [T]
```

`[M]` is the number of processes to run `game` on. If you want to have $C = 8$ worker processes, then you need 9 processes in total, accounting for the manager. Hence, you should write `-np 9` in the command line. The flag `--oversubscribe` allows you to set `[M]` more than just the number of logical cores on your machine, which we will do.

Arguments `input.txt`, `output.txt`, and `[T]` are passed onto `game` as command line arguments. The `input.txt` will contain the initial state of the map of size 360×360 with;

- rows separated by a single new line (`\n`) character,
- each cell on a row separated by a single space ( ) character,
- each cell as a `0` or `1` for emptiness and life.

The `[T]` is the number of iterations to simulate the Game. The `output.txt` should be filled with the map's final state after `[T]` iterations of simulation, with the same syntax as in the input file, described above.

2.2 The job of the manager and worker processes

MPI is flexible, meaning that it is designed to be able to accommodate parallel computation models other than just the manager-worker model. Hence, it will not designate a process as the "manager" by itself. Instead, you should regard the rank[1]-zero processes as the manager yourself in your MPI code.

The job of the manager (rank $= 0$) process:

1. File input/output.
2. Splitting the data into **completely separate (i.e. disjoint)** parts and sending them to the workers.
3. Receiving the results from the workers, and their aggregation.

The job of the worker (rank $= 1, \ldots, C$) processes:

1. Receiving the data from the manager.
2. Carrying out the simulation for T-many time steps. In the mean time, communicating with the other workers when necessary.
3. Sending the data back to manager after T time steps.

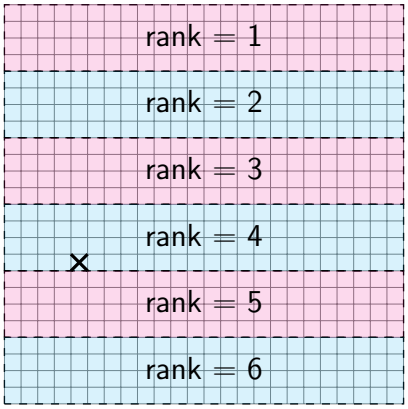
Important! The worker processes may not make any file input/output.

2.3 Splits

The maps that we will be using to test your implementation will always be 360×360. We will be using smaller square maps in this section, only to have visualizations that look better.

There are many ways to split and distribute a $S \times S$ map to the workers. We will only consider the following two:

1. **Striped:** The map will be split into C rectangular arrays, each with S/C rows and S columns. The following image shows how this split would be with $S = 24$ and $C = 6$.

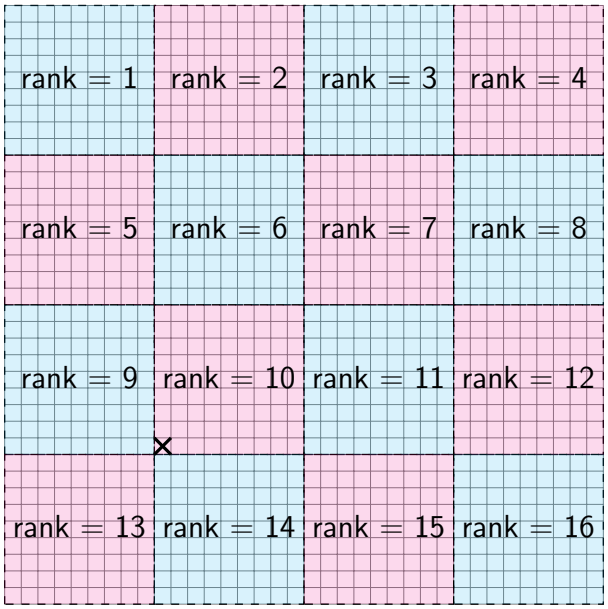


[1] The index of a process is called its *rank* in MPI.

We will refer to the cell with the black cross later in section 2.4.

If you choose this one, we will be testing your implementation with a C that divides 360. Moreover, we guarantee that C will be an even number, which should help you in implementing the communication phase efficiently.

2. **Checkered (30 pts bonus):** The map will be split into C square arrays, each with $S/\sqrt{C}$ rows and columns. The following image shows how this split would be with $S = 36$ and $C = 16$.



If you choose this one, we will be testing your implementation with a C that is a perfect square, say $C = c^2$. We also guarantee that c will divide 360. Finally, c itself will always be an even number, which should help you in implementing the communication phase efficiently.

Note In your project, please pick either the striped- or checkered-split, and implement *only* that. In other words, do not implement both of them.

Warning Splitting the map checkered is not much harder than splitting it striped. However, the checkered-split does make the communication phase (explained in the next section) very much harder! Please be aware of that before choosing to implement the checkered-split.

2.4 Communication between workers

After the split, each process will have a separate part of the map. To calculate the next state of some cells, they will need some information from each other.

For example, see the black-crossed-cell in the striped-split example. Process 4 will have to know the states of the 8 cells around it to calculate the black-crossed-cell's next state. Three of them, however, belong to the rank 5 process. Similarly, there are some cells that

rank 4 process knows about, and the rank 5 process needs to know. To fulfill their needs, rank 4 and rank 5 processes will have to communicate and exchange those information with each other. In fact, rank 4 and 5 processes will have to exchange a copy of their whole bottom and top row, respectively.

As for the checkered-split example, process 10 needs information from processes 9, 14, and 13 to calculate the next state of the black-crossed-cell. Information exchange between processes 10-9 and 10-14 is similar to the one in striped split case. Rank 10 and 13 processes will have to exchange only one cell of information, about their bottom-left and top-right cells, respectively.

If you consider all the boundary cells within a part of the map:

- With the **striped-split**, each process will have to send their *entire* top/bottom row of their part of the map to the process above/below. On the other hand, each process will also have to receive the *entire* bottom/top row of the part of the map belonging to the process above/below, respectively.
- With the **checkered-split**, each process will have to make the similar exchanges as in the striped-split case, but also with their *entire* leftmost and rightmost columns, and with the processes on the left and on the right. Moreover, each process will have to send the corners of their part of the map to the process that is above-left, above-right, etc. On the other hand, each process will also have to receive the bottom-right, etc. corners of the part of the map belonging to the process that is above-left, etc.

This exchange of information will have to be done once in every time step of the simulation.

A process can find out which processes it will have to communicate by looking at their own rank, and the total number of workers. This is relatively easy for the striped-split. See the following two formulae that gives the zero-based[2] row and column index of a process with rank $r = 1, 2, \ldots, C$ in the checkered-split case:

$$\text{Row}(r) = \left\lfloor \frac{r-1}{\sqrt{C}} \right\rfloor$$
$$\text{Column}(r) = r - \sqrt{C} \cdot \text{Row}(r) - 1$$

2.4.1 Deadlocks

Beware! You can easily have deadlocks, if you do not plan your communication structure well.

MPI does not come with spooling[3] capabilities. This means that the sender waits until the receiver receives, and the receiver waits until the sender sends.

In particular, functions `MPI_Send` and `MPI_Recv`, used in sending/receiving data to/from another process, blocks the execution from continuing until that process receives/sends the data. If two processes were to mutually attempt the same action, they would be waiting eternally.

Added 8 Dec

Important! There are other functions to send/receive in the MPI library. For this project, you may only use `MPI_Send` and `MPI_Recv` to send and receive data between processes. Using other variants (such as `MPI_Isend`) is **prohibited!**

[2] https://en.wikipedia.org/wiki/Zero-based_numbering

[3] Queueing of sent information at bay, allowing recipient to receive/process them at a later time.

2.4.2 Performance

Your algorithm must be parallel and performant. We will readily tell you how you can achieve the required performance by the end of this section. Please implement your code accordingly, and otherwise, you may lose some points.

Some communication schemes are more efficient than the others. Considering the striped-split's example, an example communication scheme that implements periodic boundaries without causing any deadlocks would be:

- Process 1 sends its bottom row to process 2. Then, it starts *waiting* to receive a row of information from process 6.
- Process 2 receives from process 1. Then, sends its bottom row to process 3. Process 3, 4, and 5 performs similar to the process 2.
- Process 6 receives from process 5. Then, sends its bottom row to process 1.

Unfortunately, with this scheme, there is a waiting chain of 6 communications for the process 1 to receive its information. Not just the process with rank 1, but also any other process x will have to wait for the previous $x - 1$ communications to be completed, before receiving the information.

We can say that this communication scheme performs in $O(C \cdot \sqrt{n})$ time, where n is the number of cells on the map. This means that each added worker comes with a time penalty.

Now, note that the following naive "simultaneous send" communication scheme would not be better, and in fact cause a deadlock, and therefore would not work:

- Every process sends its bottom row to the process below. Every process then waits to receive a row of information from the process above.

Fortunately, there still is a way to achieve $O(\sqrt{n})$ time for the communications. We cannot make *all* of our processes send at once; but we can make every other process send, while the remaining ones are waiting to receive. Since we are assuming that C (or $\sqrt{C}$ in the checkered-split) will be even, we can partition our processes as even/odd by their ranks, and establish the following scheme:

- Every process with an odd rank sends its bottom row to the process below. Then, they wait to receive a row of information from the process above.
- Every process with an even rank receives a row of information from the process above. Then, they send their bottom row to the process below.

It is very much important to us for you to notice the performance problem in the sequential scheme, the deadlock problem in the "simultaneous send" scheme, and how the even-odd scheme improves and achieves parallelism. You are welcome to implement your own ideas that runs as efficient. On the other hand, you will lose some points for implementing an inefficient communication scheme.

3 Task Outline

Write your code **in C or C++ language**, implementing the algorithm that simulates the Game of Life as described above.

Here is the list of tasks you will have to accomplish during this project, not necessarily in this order:

1. Get familiar with the MPI framework. Play with the examples that are bundled with the Open MPI release distribution. You will have to know about the following functions at the very least. These are also *enough* to complete this project:

   Changed 8 Dec

   - `MPI_Init` and `MPI_Finalize`
   - `MPI_Comm_rank` and `MPI_Comm_size`
   - `MPI_Send` and `MPI_Recv`

   See the documentation in the following link for the documentation of the Open MPI library to find about those functions and more:

   https://www.open-mpi.org/doc/current/

   Added 8 Dec

   You will see that there are $6 + 1$ different "send" functions available in the documentation. However, for this project, you are **not allowed** to use any other function than `MPI_Send` and `MPI_Recv` to send and receive messages between processes.

2. Write the code that will make the rank-zero process act as the manager, as described in section 2.
   (a) Read from the input file.
   (b) Distribute the map either with *striped-* or *checkered*-splits, as described in section 2.3. Recollect the map from workers.
   (c) Print the map to the output file.

3. Write the code that will make the higher rank processes act as the workers, as described in section 2.
   (a) Receive a part of the map from the manager.
   (b) Send and receive the missing cells to and from other workers. Evolve the map for one iteration. Handle the evolution of the boundary cells via *cutoff* or in the *periodic* manner, as described in section 1.2. Repeat this for T-many iterations.
   (c) Send the part of the map back to the manager.

Implementing the periodic boundaries and the checkered-split is not necessary to get the full credit. If you implement the periodic boundaries, you will get a **20 pts** bonus. If you implement the checkered-split, you will get another **30 pts** bonus.

We will have a demo session shortly after the submission deadline. Each of you will explain how your MPI code accomplishes the given task, and demonstrate it work. The demo is mandatory for getting any credit.

4 Submissions

The project is due December 20 at 17:00, strict. Make your submissions to Moodle by then, following these guidelines:

1. Write a short report of your program that explains your design decisions, reasonings, and assumptions. Please follow the Programming Project Documentation guidelines. Some sections (e.g. "Program Interface"), despite being given here, must also be written into your report with your own words. This is for a good practice.

2. Your code should be self explanatory by means of either choosing self-explanatory variable/function names or explicit comments.

3. Comment your name, your student id, compilation status (Compiling/Not Compiling), Working status (Working/Not working) and any additional notes at the beginning of your main source code file.

4. Your submission must be your own work. Any similarity with another submission will be considered as cheating.

5. Submit your deliverable as a zip file that includes your report and implementation codes in separate folders in Moodle, until December 20, 17:00. You do not need to submit any hard copies of your deliverables.

6. Note that the deadline is strict and late submissions will be affected by the late date policy of the course.