MA/CSSE Homework 7
Due 5/12

Directions

- **Each problem must be self-contained in a file**, named properly, and **must compile using the command** `mpicc filename.c`
- Turn in each `.c` file to the dropbox. **Do not create a `.zip` file**
- On this homework you may not use any of the standard default MPI collective operations.

Problems

Level: Easy

1. Create a file called `bucketsort.c` that implements the bucketsort algorithm discussed in class, and that is represented in figure 4.10 (page 120) of the book. Basically, the algorithm has the following steps:
   (a) The root processor reads in the data to sort.
   (b) The root processor divides the initial list, L, into p sub-lists, where p is the number of processors. Let's call the i^{th} sublist L_i. Each L_i is of length approximately n/p. The root processor distributes L_i to processor i.
   (c) Processor i divides L_i into "sub buckets", $B_{i,j}$.
   (d) Each processor i sends all of its sub-buckets B_{ij} to the corresponding processor, j.
   (e) Each processor uses quicksort to sort its bucket.
   (f) Each processor sends its data back to the root node to be appended to the final master list.

   The file `bucketsort_start.c` provides a start for you, and the file `bucketsort_helpers.h` provides some utilities for parsing the command line, printing usage, etc. Note that `quicksort` is provided for you.

   You may compile the standard code using:
   `$ mpicc -lm objs/bucketsort_standard.<arch>.o -o bucketsort_standard` Notice that you need to link the math library using `-lm`.

   Here are the criteria to keep in mind:
   (a) You must produce the correct results. Respect the `--print` option, it is stored in the `print_things` field of the options struct. If the sorted list is to be printed, print it out in the exact same format as the standard code.
   (b) Only the root processor should ever have a full copy of the list. So, only it should read the input file if one is provided, and only root 0 should generate the random data if there is no input file provided.
   (c) You must produce a file containing the sorted list – this is taken care of with `write_outputfile`
   (d) You must use `MPI_Scatterv` to do the initial distribution of data from the root, `MPI_Gatherv` to do the final collecting of data back to the master, and `MPI_Alltoallv` for the swapping of "sub-buckets". Besides those communications, your other messages may total no more than $10p$ integers.

(e) Your code will be run with $1, 2, 4, 8, 16$, and 32 processes. Your code may take no more than 2 times as long as the standard code in each case. As with homework 6, you may skip some of this testing using the `--max-proc=<n>` option to the `test_bucketsort.py` program.

Level: Difficult

Complete the problem above. In addition:

1. *Cellular Automaton* are a classic construct widely used in mathematics in computer science. The have been used for strictly to model complex phenomena like traffic flow, spread of forest fire, population dynamics, and crime incidents in urban areas. The most famous cellular automaton was developed by John H. Conway in the 1960's, and is called the *Game of Life.* Canway developed the game to illustrate that complex behavior can arise out of simple systems using only simple rules.

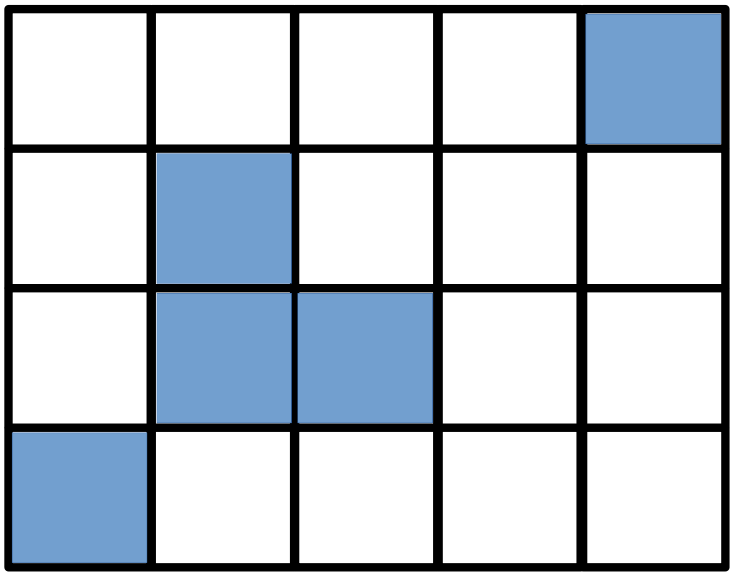
   Cellular automata consist of an initial grid of cells, each cell may be in one of finitely many states. The initial set of states is called the *initial condition* of the cellula automata. From the initial condition, the system progresses, from one *generation* to another, following a prescribed set of rules.

   In the Game of Life, we imagine that an organism may occupy each cell. Therefore each cell has only two possible states, empty or occupied. Each cell has eight neighboring cells. The rules to transition from one generation to another are:

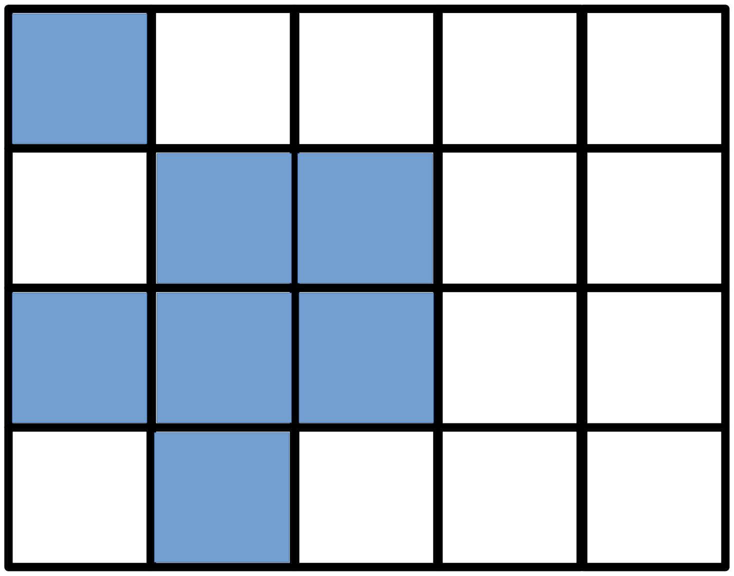
   - Any organism with two or three neighboring organisms survives for another generation.
   - Every organism with four or more neighbors dies from overpopulation.
   - Every organism with one neighbor or none dies from isolation.
   - Each empty cell adjacent to exactly three occupied neighbors will give birth to an organism.

   We assume that the board "wraps around".

   For example, imagine that this were the initial state of a GOL board:

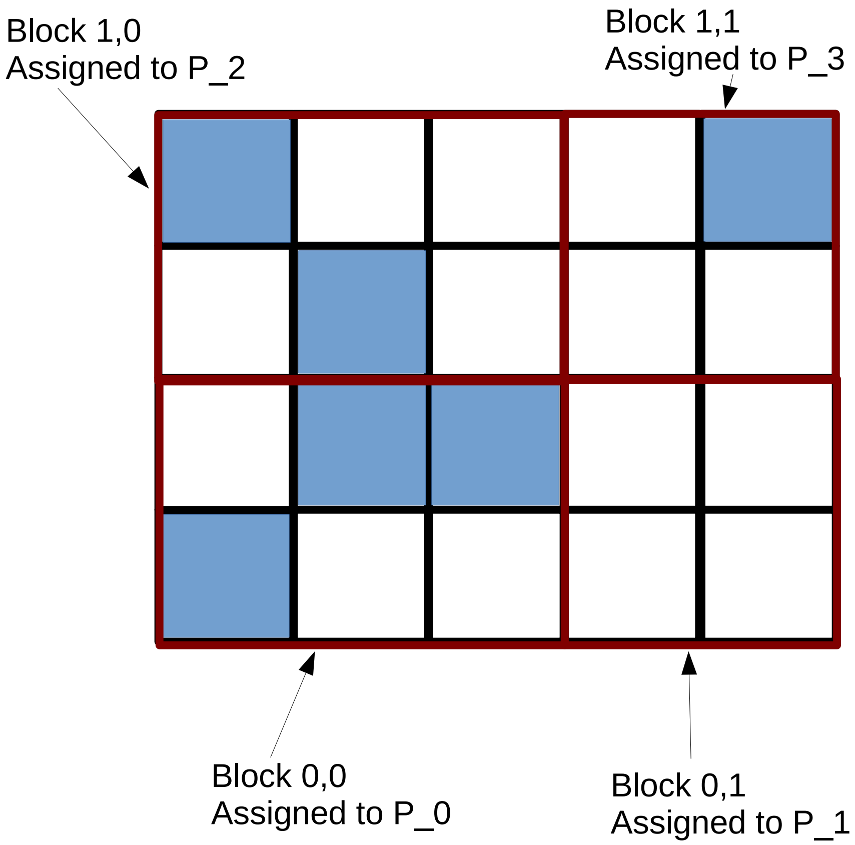


   Following the rules above, the next generation of the board is:



In the file `gol.c` write a program to play the game of life using many processors. The code in `gol_start.c`, `gol_helpers.h`, `gol_comm_helpers.h` provide the code to read and set the appropriate simulation options, utilities for producing the output gif, and tools for use during the simulation.

Suppose the grid is m rows by n columns, and that we have p processors to use. In the figure above, $m = 4$, $n = 5$. Imagine overlaying the grid with a *block grid* of dimensions $M \times N$, and assigning each processor to update only the cells that lie within its assigned block. In the figure below, $M = 2$, $N = 2$.



We can see that in order to update the cells in its block from generation t to $t+1$, each processor must receive information about the cells on its border from its neighboring processors. Similarly it must sent the state the cells on its border to all neighboring processors so that they may update their generation.

Here are the facts you need to know about the simulation:

- The number of generations in `options` includes the initial generation that is read in from file.
- The board wraps around in all directions, there is a function `get_state` in `gol_helpers` that will get the state of a board at generation t, row i, column j. It handles wraparound properly, so that asking for row 1, column -1 will get the *rightmost* entry of row 1.
- The state of a cell i, j at time t can be set using `set_state`. You **must** use `set_state` to set the state of the board every time – it is how I keep track of how much work each processor does.
- All indexing is 0 based.
- Determining the dimensions of the block grid, and assigning blocks to processors is done for you.
  - In `gol_comm_helpers.h` the function `get_block_dims` returns the block dimensions M and N to use on a board with dimensions m and n when using n_{procs} processors.
  - `get_block_coords` returns the coordinates (block row, block column) of the block assigned to processor `rank` if the block grid is $M \times N$.
  - `block_coords_to_rank` returns the rank of the processor assigned to block b_i, b_j (block row, block column) if the block grid is $M \times N$. This handles wraparound correctly, so asking for the processor assigned to $-1, 0$ returns the processor assigned to $4, 0$ if the block grid has 5 rows.
  - `get_assignment` returns the leftmost column, rightmost column, bottommost row and topmost row (all inclusive), that the processor with block indices b_i, b_j should work on if the board is $m \times n$ and the block grid is $M \times N$.

- At each generation, each processor must share only the correct amount of data (the number of cells on its perimeter) with its neighboring processes. Each processor should send its boundary data to itself, if it is its own neighbor (simplifies the coding a lot).
- In order to reduce the potential for deadlock, you must use the `MPI_Sendrecv` command to transfer all boundary data.
- At the end of the simulation, every processor must send its complete information back to the root processor. The root processor must then produce an animated gif of the game.
  - The function `copy_board_range_to_buffer` copies a specified section of the board into a contiguous buffer suitable for sending, it handles wrapping around properly.
  - The function `copy_buffer_to_board_range` copies a buffer to a specified chunk of the board, it also handles wrap-around correctly.
- The files `gol_serial.c` and `gol_serial_helpers.h` give you a complete serial implementation of the game. The files `gol_start.c` gives you a start on the parallel version, and `gol_helpers.h` and `gol_comm_helpers.h` are the appropriate helper functions.
- It is perfectly reasonable on this assignment to modify my helper files, or to create your own helper files. So, you **must submit all files required to compile your code, including the ones I distribute to you**. Your code must compile with a simple
  `mpicc gol.c` Feel free to modify any of my functions that you want, with the exception of `set_state`. You do not need to submit any of the initial condition files, etc.
- Your code will be run with $1, 2, 4, 8, 16$, and 32 processes. Your code may take no more than 2 times as long as the standard code in each case. As with homework 6, you may skip some of this testing using the `--max-proc=<n>` option to the `test_gol.py` program.
- You may use `rang_gol_board.c` to create random initial states if you like.