Assignment 9 - Documentation

July 10, 2019

Contents

1 Conway's Game of Life

Conway's Game of Life is a popular cellular automaton. The game is played on an infinite 2-D grid where each cell is either dead or alive. In each iteration of the game, we apply the following rules to determine the next state of the grid:

1. An alive cell with fewer than two live neighbours dies, as if by under-population.
2. An alive cell with two or three live neighbours lives on to the next generation.
3. An alive cell with more than three live neighbours dies, as if by over-population.

4. A dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

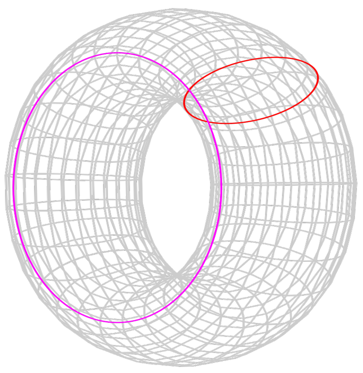
For more details on how the game works, and interesting patterns that have been observed, go read the Wiki.

2 Implementation Details

2.1 Game of Life on a Torus

In life_seq.c, you will find a sequential implementation of Game of Life. Since we do not have infinite memory, in our sequential implementation we instead use a 2d torus - a 2d grid with its columns and rows wrapped around: imagine taking a really wide sheet of paper, joining the top and bottom together to get a cylinder and then gluing the edges of the cylinder together.

The game of life rules amount to applying a 9 point stencil to each point - the next state of each point depends on its eight neighbours, left right, top bottom and the four diagonal neighbours. Since the rows and columns wrap around in our implementation, points on the edges of the 2d grid are connected to each other.



https://upload.wikimedia.org/wikipedia/commons/8/81/Torus_cycles.svg

In order to acheive this in the sequential versiono, we pad our 2D grid with additional rows and columns - one additional row on the top and one additional row on the bottom. Likewise, we have one additional column on the left and one additional column on the right. The height and width variables in life_seq.c include these padding rows and columns. These additional rows and columns contain copies of the values on the edges of the grid.

For more details on the implementation, refer to the evolve function in helper.c.

2.2 Parallelization Considerations

We want to use MPI Collectives to parallelize the sequential version of the code. In order to do this, we have to think about the following things:

1. How will we decompose our domain?
2. How can we distribute data between different processes?
3. Do we need to change any of the code from the sequential version?

One of the first ideas you might have for the decomposition might be similar to the assignment 8: decompose along the rows and send multiple rows to each process. This is a good strategy and you should consider using it. However, you are welcome to try any other strategy you might want. Additionally, you might want to look into using MPI's virtual topologies, like a cartesian grid, to help you acheive this. However, this is certainly not required.

To distribute data between different processes, you might want to employ the ghost cell pattern. This can be quite helpful for iterative stencil codes, such as the Game of Life or a Jacobi solver, where the at the boundary of each domain, we need data from another process. To do this, each process allocates additional space for data that it needs from neighbouring processes - we call these ghost cells. For more details, please refer to this excellent introduction .

To answer the third question, depending on how you parallelize your code, you might need to rewrite the evolve function from helper.c since it does not take into account any domain decompositions that you might perform. However, you should still be able to use most of the core logic, namely the game of life update rules, that this function implements.

3 Running your code

3.1 Normal Operation

As with the first assignment, please try to run your code with the following signature:

```
mpirun -np <num procs> <path to executable>
```

There are functions in helper.c, especially "save_to_file" that you can use to debug your code.

3.2 GUI

This assignment also comes with a GUI. To launch the GUI, you can use the `-g` option to launch a window which displays the simulation in real time.

```
mpirun -np <num procs> <path to executable> -g
```

While it is not required for this assignment, you can also do this for your parallelized code. To do this, you have to call `gui_draw` whenever you want to draw a frame. See the sequential code for an example. The flag to draw the gui (`global_draw_gui`) will only be set in the main process, and only if the command-line option has been specified.

If you cannot compile your code due to issues with X11 or OpenGL, you can try building it with

```
make NOGUI=1
```

This will disable any functionality related to the gui.

4 Speedup requirements

Your parallel code, without the GUI, should be able to acheive a *minimum* speedup of **13** with **16 processes**.

5 Assignment Deadline

The deadline for this assignment is on the **16th of July at 23:59.**