Assignment 11 - Documentation

July 10, 2020

Contents

1 Conway's Game of Life **2**

2 Implementation Details **2**

2.1 Game of Life on a Torus 2

2.2 Parallelization Considerations 3

3 Running your code **4**

3.1 Normal Operation 4

3.2 Video Output 4

4 Speedup requirements **4**

5 Assignment Deadline **5**

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 relevant article on Wikipedia. Wikipedia also contains some interesting information on some patterns that occur in the simulation, which is potentially interesting for some optimization.

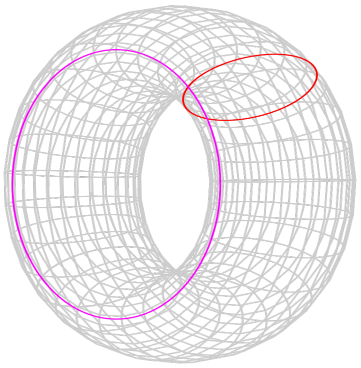
2 Implementation Details

2.1 Game of Life on a Torus

In sequential_implementation.cpp, you will find a sequential implementation of the 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.

In order to acheive this in the sequential version, 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



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

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.

The main game of life logic is implemented in a function called evolve.

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 last assignment, splitting along the rows and sending one or more rows to each process. This is a good strategy and you should consider using it. Note that this time around, the number of rows is not neatly divisible by the number of processes.
However, you are also 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 a more advanced topic and certainly not required.

To distribute data between different processes, you might want to employ the halo/ghost cell pattern. This can be quite helpful for iterative stencil codes, such as the Game of Life or a Jacobi solver, where 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 neighboring 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 also modify some functions from Utility.cpp since it does not take into account any domain decompositions that you might perform. To do this, copy the function over to your code, rename it and modify it.

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>
```

Both Utility.cpp and VideoOutput.cpp contain some useful methods to help you visualize/analyze what your code is doing.

3.2 Video Output

This assignment also comes with an improved version of the Video/Picture Output. To activate it, you can use the `-g` option to write video and image output. The video output requires FFMPEG to be available on your path. In this improved version, the video should now be viewable in more media players. As an alternative, it is now also possible to export single frames to PNG. This should no longer require additional dependencies.

```
./sequential_implementation -g
```

While it is not required for this assignment, you can also do this for your parallelized code. To do this, call the relevant video output functions only on rank 0.

Make sure to conduct any timing experiments with visualization disabled.

4 Speedup requirements

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

5 Assignment Deadline

The deadline for this assignment is on the **20th of July at 23:59.**