Conway’s Game of Life

Mark Mazel (mm4764)

November 2023

1 Introduction

Conway’s Game of Life is an example of a cellular automata algorithm. It is run over a 2D bit grid where each 1 bit represents an ”alive” cell and each 0 bit represents a ”dead” cell. Which cells are alive and which are dead in each generation depends purely on 9 bits from the previous generation: the cell itself and its 8 direct neighbors. These are governed by the following rules (from Wikipedia):

1. Any live cell with fewer than two live neighbours dies, as if by underpopulation.
2. Any live cell with two or three live neighbours lives on to the next generation.
3. Any live cell with more than three live neighbours dies, as if by overpopulation.
4. Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction

Because of the trivial dependency of state of a given bit from the current generation to the next generation, this seems to be quite a parallelizable algorithm. It also has the benefit of being visual and (time permitting) I hope to either make a console visualization for it or dump the outputs into a file and make a JavaScript or python-based visualization for it.

2 Serial Implementation

While researching this topic I have seen multiple implementations of the algorithm in Haskell as well as other algorithms. The approach is quite straight forward - iterate through each bit and through each of it’s neighbors counting the neighbor count, check the four rules mentioned above and decide the next generation state of the cell based on that. The data-dependency of the algorithm can also be tuned as needed by running more and more generations. If the time necessary for reading the data is too large relative to the computation time, one can just run the algorithm for more steps. Likewise, the problem can be made harder by scaling the size of the grid (and there are also more complex versions of the automata setup which treat the 2D grid as being mapped over a torus).

3 Parallel Implementation

The parallel approach I would like to try with this is splitting the grid into smaller grids and computing each one in parallel. This will give some room for tuning these parameters for better performance. Another portion here which can be tuned is how the edges between these grids are handled. For example, the upper leftmost cell in a sub-grid will require the values of 5 cells not contained in the sub-grid. Some possibilities I can think of are doing the edges as separate sub-grids, using sub-grids with a larger size of data than the grid they will generate as an output, or having the computation on one sub-grid pass on to its adjacent grids some supplemental data of this adjacency (however I fear that this will create unnecessary bottle-necking/serialization of the parallel computation).