

Game of Life: Project Proposal

Casey Roman (cr3324)

Parallel Functional Programming — October 2026

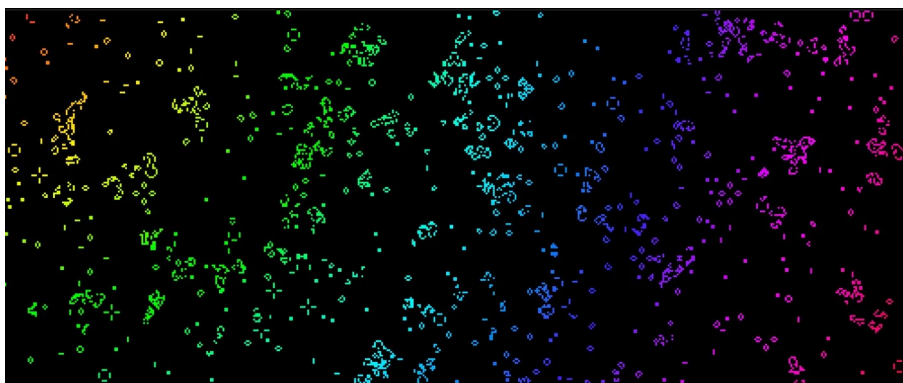


Figure 1: A rather aesthetic instance of the Game of Life.

1 Introduction

John Conway’s Game of Life (GoL) is a cellular automation devised in 1970. It consists of an infinite grid of cells, each cell in one of two states: alive or dead. At every generation, cells update based on their eight attached neighbors (their Moore neighborhoods) based on the following rules: a live cell lives on with two or three live neighbors, and a dead cell becomes alive with three. It is a zero-player game, so its evolution is decided by the initial state (seed) and requires no further input. Despite these very simple rules, runs of the game can create very interesting patterns, as pictured in Figure 1. It is a Turing Complete problem.

2 Motivation

I learned about the GoL years ago, and it has stuck with me ever since. I am charmed by the seemingly simple, manual upbringings of the game, the many unique ways it has been used since, and its association with life, hence the

name. There are a lot of reasons why I like it, but I will not get into all of them here.

Thinking more practically, the GoL is a very straightforward algorithm and its core. It is the vastness and mystery of its patterns that make it interesting. Because each cell requires little computation, simulating the game is limited by memory, thus making it a good target for optimization. Of course, the GoL has been optimized in many ways, but Haskell provides a new canvas to implement and improve some popular algorithms that speed up the game.

3 Approach

3.1 Sequential

I intended to translate a known, somewhat optimized sequential algorithm for the Game of Life. As aforementioned, there have been many efforts for this already, and I must do some more research to determine what would best for the scope of the project. A common optimized version I found involved using two arrays: one for the current state and a second for the current state. Each generation flips the arrays to avoid wasting time and space to make an unnecessary copy. This can be implemented in Haskell using unboxed mutable arrays. Unboxed allows the cells raw data value to sit side by side in memory, saving both space and time. There is less garbage to collect, unlike lazy boxed arrays. Mutable arrays allow us to replace each cell individually (instead of the entire array, aka our grid.) This allows us to reuse the two arrays. Clearly, this seems to contradict the immutability aspect of functional programming, so we must leverage the ST monad that allows mutation of local variables and arrays. Again, more research needs to be done, but this sequential approach seemed doable.

3.2 Parallel

Since each cell's next state only depends on the current state, different parts of the grid can be computed in parallel. For each generation, I can divide the grid into pieces, compute them in parallel, and once they've completed, compute the next generation the same way. We can divide and conquer in a couple ways, namely "strips" of the grid or "tile" of the grid. We will be able to tune the size of both strips and tiles.

With tiles, we can consider the "active tiles" method that only computes tiles that changed in the previous generation or are touching ones that did. This works because all other tiles are guaranteed not to change. If this initial parallelization is successful, we can continue by adding load balancing. Due to the variable nature of the grid, some pieces may complete before others. If we are able to pass on more work to threads that complete early, we can create a potentially faster program. From my basic research (since we haven't gone over parallelization in class yet), strategies and the Par monad seem most applicable.

4 Input Data

Input data shouldn't be too hard to obtain since the GoL has many well researched patterns to use as a baseline. These standard patterns can be used to ensure the program is working as it should. As for scaling, I can use random seeds to create unique grids of varying sizes for benchmarks. There are also many patterns available online I can utilize.

References

- [1] **Inventing Game of Life (John Conway) - Numberphile**
<https://www.youtube.com/watch?v=R9Plq-D1gEk>
- [2] **Conway's Game of Life - Wikipedia**
https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life
- [3] **How I optimized Conway's Game Of Life - Medium**
<https://medium.com/tebs-lab/optimizing-conways-game-of-life-12f1b7f2f54c>
- [4] **Arrays - HaskellWiki**
<https://wiki.haskell.org/index.php?title=Arrays>
- [5] **Monad/ST - HaskellWiki**
<https://wiki.haskell.org/Monad/ST>
- [6] **monad-par - Hackage**
<https://hackage.haskell.org/package/monad-par>
- [7] **Control.Parallel.Strategies - Hackage**
<https://hackage-content.haskell.org/package/parallel-3.3.0.0/docs/Control-Parallel-Strategies.html>