Ixil D. Miniussi
31124607
idm1u19@soton.ac.uk

Conway's Game of Life implemented in POETS (draft)

Project description

The goal of this project is to implement Conway's Game of Life[2] within the POETS framework, each cell being handled by a separate node. Before going into the POETS implementation, a quick rundown of Conway's Game of Life.

Conway's Game of Life

Background

"Conway's Game of Life", also known as "The Game of Life" or "Life" is a cellular automaton from British mathematician John Horton Conway. The game has no goal or objective, and some people would say it also doesn't have a player, since the evolution of the game is fully determined from the first generation. The user's only agency is to devise the initial state depending on whichever goal he sets for himself. Since its initial conception, the game has been shown to be Turing complete[1], and users have been able to devise such programs within the game as logic gates, clocks, or even The Game of Life itself.

Rules of the game

In its traditional form, The Game of Life is played within an infinite 2d grid, with each cell of the grid holding one of two values: alive, or dead. Each generation, the cells of the grid update themselves simultaneously based on their current state and that of their neighbours (horizontally, vertically, and diagonally adjacent cells):

- A live cell with less than two live neighbours dies.
- A live cell with two or three live neighbours lives on.
- A live cell with more than three live neighbours dies.
- A dead cell with exactly three live neighbours becomes alive.

There are semi popular alternative rules to the game, one of them imagining using hexagonal cells to increase the amount of neighbours, and we will consider exploring these later down the line.

Goals of our implementation

There are two ways of measuring a successful implementation.

The first is functionality. As explained, the game of life is fully deterministic, which means separate runs of the same generation will always lead to the same result. Our implementation must be no different, and must lead to the same results as any other correct implementation of the game.

The second part is scalability and speed. The purpose of the POETS platform is to handle massively parallel computations. Our implementation must make full use of the platform and avoid inefficient bottlenecks, and be capable of handling a large implementation of the game.

POETS implementation

One of the main challenges of our implementation is to make full use of event-based computing and avoid bottlenecks.

The first obvious decision comes with handling each cell in its own POETS device, there can be a great many of them handled at once and we wouldn't be making full use of the POETS platform any other way. This unfortunately comes at the cost of changing the rules of the game slightly, as we can know longer virtualize a fully infinite grid, and will instead opt for a wrapping grid (as other implementations sometimes do).

Then, when handling iterations it is tempting to wait for all cells to be ready before moving on to the next generation. This means having an aggregator (most likely a supervisor) collecting from and notifying all cells each update. This would create a bottleneck which wouldn't scale well with the problem size. More importantly, our program does not need to be globally synchronized in order for it to be functionally correct. As such here is the implementation:

Device/Cell algorithm

Each Cell is represented by a POETS device. The device retains the following information:

- *bool alive* - Holds true if the cell is alive, false otherwise.
- *bool even* - Holds whether it is on an even or uneven generation.
- *int generation* - Increases with each new update.
- *vector[int] conversations* - Holds 2 values counting the amount of conversations so far.
- *vector[int] livingNeighbours* - Holds 2 values counting the amount of living neighbours so far.
- *bool sendMessage* - Holds true if it wants to send a message.

Each node begins by sending a message called "small-talk" to its neighbours. Within each small-talk is contained the current living state and generation (whether it is even-numbered). When receiving a small-talk, the cell looks at whether it is coming from an even or uneven generation.

Algorithm 1 onReceive

if small-talk is from an even-numbered generation **then**
    increment conversations[0]
    if the talking cell is alive **then** also increment livingNeighbours[0]
    end if
else
    increment conversations[1]
    if the talking cell is alive **then** also increment livingNeighbours[1]
    end if
end if

if the listening cell is on an even generation **and** conversations[0] == the total number of neighbours (currently 8) **then**
    apply the rules using livingNeighbours[0]
    set even to *false*
    set sendMessage to *true*
    reset conversations[0] to 0
    reset livingNeighbours[0] to 0
end if
if the listening cell is on an uneven generation **and** conversations[1] == the total number of neighbours (currently 8) **then**
    apply the rules using livingNeighbours[1]
    set even to *true*
    set sendMessage to *true*
    reset conversations[1] to 0
    reset livingNeighbours[1] to 0
end if

Design decisions

Why vectors?

The reason for using vectors is to handle the asynchronous nature of our algorithm. We want it to work using local synchronization without losing the functionality of global synchronization.

A scenario we are looking to avoid goes as follows:

- Cell 1 (gen 1) sends small-talk to Cell 2 (gen 1.)
- Cell 1 (gen 1) receives small-talk from Cell 2 (gen 1) and all its other neighbours (gen 1), and changes state (gen 2.)
- Cell 1 (gen 2) sends small-talk *again* to Cell 2 (gen 1) before it has yet to update.
- Cell 2 (gen 1) receives small-talk from Cell 1 (gen 2) and all its other neighbours (gen 1) and updates into generation 2 using states from both generation 1 and 2, breaking the determinism of the game.

For that reason, we want to store the corresponding generation from each message. However, once we start storing said information, we notice that no two neighbors can be over one generation apart, since one cell necessarily needs its neighbours to be at least on the same generation before updating.

Knowing this, we only need to store 2 data points maximum per neighbouring cells. One way of doing this is by thinking of generations as even or uneven, and storing them accordingly in our *conversations* and *livingNeighbours* vectors.

Livelocks and Deadlocks

On a micro level, POETS ensures that every single message eventually arrives at their destination, and as long as messages are being received, cells update and send new messages. As such there should be no deadlocks on a micro level.

On a macro level, there are certain iterations to The Game of Life which ends in repeating patterns or complete livelock. We are not looking to address these. More naturally, the user himself specifies how many generations he wants to run when running the simulation (P). Once cells reach that generation, the program ends.

Requirements

Functional

The Game of Life is a fully deterministic game. The player has agency with how the first generation is laid out, but the unfolding ought to be the same with every run in every implementation of the rules. This version should be no different. A simple but time-inefficient way of verifying this requirement is to go through a run, and compare every iteration with a reference implementation.

To test more scenarios in less time however, we can also compare every C generations, by assuming that if a generation is accurate, then so were the generations leading up to it. As such, our implementation makes testing easier by letting the user choose on launch the value of C, which will affect how often iterations get rendered. These iterations will get written onto a corresponding csv file to make it easier to compare with third-party implementations.

Non-functional

Besides changing the rules, this implementation can only evolve in two directions: size and speed.

For the purpose of this documentation, we will measure problem size as the total number of cells N.

Speed can be seen either as the total number of cell updates per second $c.s^{-1}$ or the number of new generations per second $g.s^{-1}$. The relation between the two values can be written as

$$c.s^{-1} = g.s^{-1} * N$$

A successful implementation would especially excel in scalability. There is an expectation with POETS that larger problems do not suffer the same slowdowns as other platforms.

Experimentation

There are a number of assumptions that come with implementing a game like this on a platform meant for massively parallel computing. These assumptions relate to its supposed scalability and speed.

Cheat Sheet

- N - Problem size, i.e. total number of computed cells.
- P - Total generation count, user set.
- C - Rendered cycles, every C generations, a generation is written to file.
- $c.s^{-1}$ - Cell updates per second.
- $g.s^{-1}$ - Generations per second. Due to the asynchronous nature of the algorithm this can be a little vague, but it is calculated as an average, i.e. P divided by the execution time T.
- T - The execution time.

Problem

In particular we want to know:

- How large can N be before running out of memory?
- How high is $g.s^{-1}$ at varying problem sizes? does $c.s^{-1}$ remain constant or increase through it all?
- How do $g.s^{-1}$ evolve with fixed N but varying P, specifically at lower P values.

Methodology

Regardless of the test subject, to avoid using unnecessary computer power we set C to P as to write to file only the starting and final states.

- To test the maximum problem size, we increase N progressively at a fixed but low P while testing that functionality does not suffer. We also look for any sudden and/or abnormal decrease in $c.s^{-1}$.
- To test scalability, we pick a fixed and relatively large P (500). We use an in-code implementation to test the total execution time T [[*could timeset work within POETS?*]] and get the average $g.s^{-1}$ through $g.s^{-1} = \frac{N}{T}$, and $c.s^{-1} = g.s^{-1} * N$. We repeat at varying N values and plot them accordingly.
- Finally, to get a closer look at the effects of smaller P values on $g.s^{-1}$. We keep a fixed problem size and measure $g.s^{-1}$ with P evolving in the 1 to 50 range.

Assumptions

Prior to testing, the maximum problem size is expected to be or be close to the maximum amount of devices supported by POETS. Since each cell is represented by exactly one device, one would assume that to be the limit.

Scalability is expected to be optimal but not perfect, meaning $g.s^{-1}$ would slightly decrease with larger N values, and therefor $c.s^{-1}$ would increase linearly with N. This is because although each cell only concerns with local synchronization, as the problem size increases we can expect some regions to fall behind more than one generation, which even with local synchronization would slow down the global system. The higher amount of messages being transferred would also reasonably cause some strain on the $c.s^{-1}$.

Smaller P values aren't expected to show much of a discrepancy with the scalability curve. Except perhaps causing more volatile results as we would no longer be averaging large generation counts.

Results

References

[1] P. Rendell. Turing universality of the game of life. In *Collision-based computing*, pages 513–539. Springer, 2002.

[2] Wikipedia contributors. Conway's game of life — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Conway%27s_Game_of_Life&oldid=1115969544`, 2022. [Online; accessed 27-October-2022].