Conway's Game of Life Project Report - Algorithm Engineering WS 24/25

Jonas Peters, Stefan Wagner, Angelo Wieden

January 12, 2025

Contents

1 Implementation

1.1 Sequential Implementation

The general idea of our sequential implementation of Conway's Game of Life in the file `game_of_life.hpp` is as follows:
We utilized a `Grid` class to store, access and modify game states as well providing a basis for counting live neighbors. Furthermore, a `GameOfLife` class was needed to manage the simulation logic, initialization as well as subgrid operations and input / output handling, which we will get to later on.

The `Grid` class' stores the grid bit by bit in a `unsigned char* grid`, as well as grid information like `rows, cols, element_count` and `_byte_count` of `size_t`. Key functions include:

- `inline bool _get_at_bit_index(size_t _index)` and `inline void _set_at_bit_index(size_t _index, bool _val)` to retrieve or set the state of a specific cell
- `inline size_t _to_index(int _row, int _col)` to calculate the linear index of a cell
- `bool get(int row, int col)` and `bool set(int row, int col)` as public cell state retrieval and modification tools
- `size_t no_neighbors(int row, int col)` to get the number of alive cell neighbors by checking each cell in a 3x3 grid radius around the target cell

- as well as getters and setters for entire rows and columns and a way to create and set smaller subgrids stemming from a grid

One could say that the `Grid` class is the heart of our implementation, setting us up with the tools needed for the actual simulation logic of Conway's Game of Life in the `GameOfLife` class, which stores our current `state` and `next_state` as `Grid` objects, and game information in terms of `rows, cols` and `element_count` as `size_t`. The game logic is handled by the following functions:

- a group of getters and setters implementing those of the `Grid` class as an intersection for our current `state`
- `void init(std::initializer_list<std::initializer_list<size_t»&& l)` to initialize a game with the passed on pairs of alive cells' coordinates like {0,2},{1,2},... by setting the state of each cell which was passed on through its coordinate pair as alive
- `inline bool becomes_alive(size_t row, size_t col)` used to check if a cell should stay or become alive in the next game tick, based on the normal Conway's Game of Life logic which goes as follows: A cell is alive in the next tick if it has exactly 2 living neighbors and is alive itself, or if it has exactly 3 living neighbors regardless of the cell's state.
- `void tick()` to go to the next game tick by checking each cell if it should come alive, setting it accordingly, and swapping the current `state` and `next_state`
- and a list of helper functions regarding debugging, creating and setting sub-versions of `GameOfLife` and input / output handling

Lastly, to account for times when an index would wander outside of the game grid, we implemented a `inline int MOD(int a, int b) {return (a%b+b) % b;}` function, which wraps the out-of-bounds index around to the other side, ensuring seam-less edge transitions. Altogether, we are now able to able to initialize a game, and tick it forward however often we like, sequentially.

1.2 MPI-Parallel

One factor in parallelization using MPI is the distribution of the processes across the grid. We want to achieve as square a shape as possible for the grid segments processed by the processes in order to maximize the area compared to the circumference, or to minimize the proportion of messages compared to the serial computing time. To do this, we divide the processes into a grid of rows and columns. The number of processes per plate and row can be defined using command line arguments. This allows us to divide the processes optimally for different image sizes and numbers of processes.

Since each calculation step depends on cells that lie outside the respective subgrid, each process allocates a number of "ghost cells" in each dimension. These "ghost cells" are used to store the values of the neighboring processes, which are exchanged between the processes via MPI.

When dividing the processes into a 2-dimensional grid, the question arises as to how we deal with the grid points at the corners of the sub-grids. The content of the grid cells in the corners is required by the processes that are diagonally adjacent to these cells, i.e. are not direct neighbors. To send the cell content, there is therefore the option of sending a small message with the content of a single cell to the diagonally adjacent process or the cell content can be sent first to the right and left neighbors, for example, which then sends two of its "ghost cells" to the upper and lower neighbors. For the sake of simplicity, we have opted for the second option.

1.3 Input/Output Handling

Parsing a Game of Life implementation solely through hardcoding the size of the board and alive starting cells is not the only supported way of initialization, as `GameOfLife`'s class function `void initialize_from_pgm(const std::string&)` allows us to freely start from any P5 `.pgm` file by parsing the board's columns and rows, which can be read in the second line of the `.pgm` file, as well as reading the following `column x row` wide matrix of data, and interpreting them as alive cells, if the data holds any other value than 0.

As outputs, we have the options of converting the final state of the game to a P5 `.pgm` file by storing the `.pgm` subtype as P5, our matrix dimensions, and each cell of our grid as black or white (0 or 255) depending on their state through the `GameOfLife` class function `void to_pgm(const std::string&) const` or simply by printing the resulting bit-matrix using `void print()`, also from the `GameOfLife` class.

2 Testing

2.1 Introduction to the Header-Only C++ Catch Framework

The framework we used for unit-testing our serial and parallelized Game-Of-Life versions was the header-only implementation of CATCH, a lightweight and portable C++ testing framework useable in a file by including the `<catch.hpp>` header and defining the `CATCH_CONFIG_MAIN` macro, which will generate a `main` function acting as a starting point for running all test cases. This allowed us to maintain readability through its clear syntax with better test reports as well as debugging and output logging capabilities than normal, scattered `assert()` calls from the library `<cassert>`.

2.2 Sequential Test Design and Implementation

Since the sequential version is not only a standalone version on its own, as it also doubles as a base for the parallelized implementation, there was a need to thoroughly make sure that everything works as expected through a plethora of unit tests for the general functionality and edge cases alike. This is done in the `tests.cpp` file included in the project and can be executed using `make test_serial` in the console. All in all, there are 30 assertions with 8 and 22 assigned to our `Grid` and `GameOfLife` classes, respectively.

The `Grid` testing includes basic Grid operations, such as

- setting and getting cell data,
- checking the grid wraparound if a coordinate would be outside of the grid,
- as well as the ability to count neighboring alive cells.

With the `Grid` unit tests done, the program continues with the `GameOfLife` tests, covering

- initialization of states,
- general Conway's Game of Life cell birth and death logic,
- a few well-known configurations such as stable and oscillating configurations,
- and edge-wrapping.

The detailed test results will be found in the `build/tests_serial/output.txt` file.

2.3 Parallel Test Design Using MPI

Parallelized testing was done through `tests_mpi.cpp` in the project folder and can be run by executing `make test_mpi` in the console. This time, there is not much of a need to re-test all the things that worked in the serial version, as both `Grid` and `GameOfLife` classes still hold true to the main integrity of the build. However, the MPI version provides the challenge of processes needing information regarding their boundary sub-grid cells. Due to this, the main importance of the parallelized tests was that the processes can properly exchange data with each other. Therefore, we opted for a four-processes, non-square grid test featuring communication by letting a glider fly through specifically the middle of a 2x2 sub-grid layout where the most boundary information is needed, ensuring that the `exchange()` function of our `MPIProcess` class works as intended.

It is worth mentioning that the tests done do not create `.ppm` files, as simply working with the binary data of the game-states is enough to prove that both the serial and parallelized versions work without issues, while being easier to implement and debug.

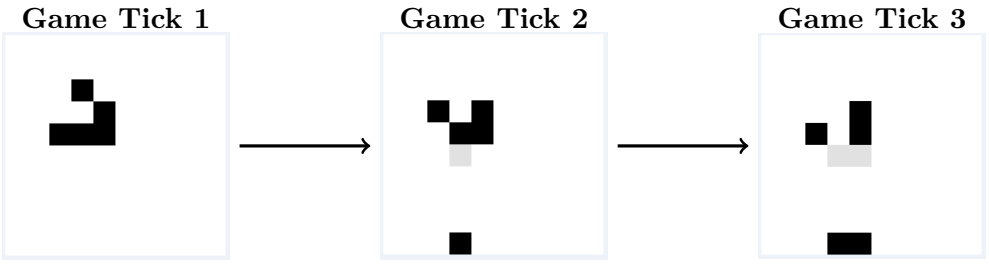
Similarly to the prior serialized tests, the detailed test results will be found in the `build/tests_mpi/output.txt` file.

2.4 Testing Hurdles and Feedback

As opposed to the serial unit tests, which were working flawlessly from the start, we had some struggles using CATCH as our testing framework on the parallelized version. This is the case as MPI requires being initialized in the `main` function of the program. However, as mentioned beforehand, the `CATCH_CONFIG_MAIN` macro automatically generates said `main` function. This problem was able to be worked out by defining the `CATCH_CONFIG_RUNNER` macro instead, allowing us to define our own `main` function and initializing MPI before manually starting the unit tests through `Catch::Session().run();`.

However, this comes at the cost of all 4 processes reporting their CATCH session at the end, while only the master process being of importance.

These test cases helped us find a bug in the parallelized version, which caused alive cells to sometimes get moved to the outer edges of the grid after the MPI processes exchanged data with each other in sub-grid bordering scenarios.



**light grey pixels show the supposed locations of cells*

This alone highlights the importance of extensive and thoroughly put together test cases in program development, as the bug could have otherwise been easily overlooked.

3 Scaling Study

In the weak scaling study, we found an almost linear relationship between runtime and the number of processes. The serial runtime is 12053.2 s. With 8 processes, the runtime was only 1517.32 s and with 32 processes only 382.314 s.

For the strong scaling study, the time is limited to 5 minutes and the number of "ticks" processed in each case is displayed. One process creates 2 "ticks", 8 processes 18 "ticks" and 32 processes "82" ticks. This increases both the workload and the number of processes.