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

.....

1.2 MPI-Parallel

.....

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,
- 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 communicate 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.

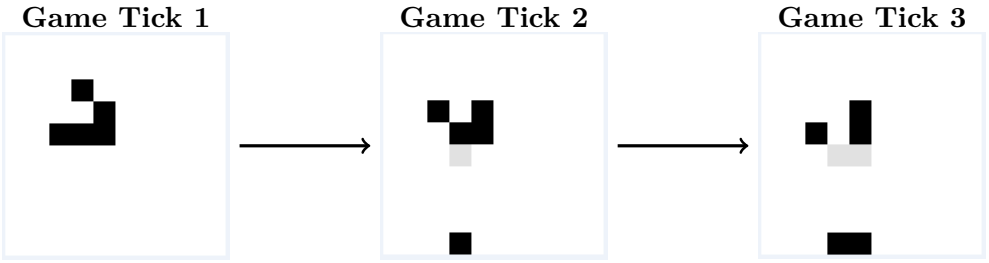
Similarly to the prior serialized tests, the detailed test results will be found in the `build/tests_mpi/output.txt` file.

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

3.1 Placeholder

.....