Conway's Game of Life

Seppe Staelens

March 28, 2024

1 Introduction

*This text constitutes a report on my version of Conway's Game of Life, in the context of the course "*Advanced Research Computing*", as part of the MPhil in Data Intensive Science at the University of Cambridge. The introduction and examples are based on the Wikipedia article.*

Conway's Game of Life is a famous zero-player game, i.e. a game where only the initial state is determined by the user and the subsequent evolution is fully determined. It is played on a (binary) grid, and each grid cell is either "alive" or "dead". Every cell has eight neighbours (diagonal counts as well), and every iteration, the grid is updated according to the following simple rules:

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.

The goal of this project was create a programme that implements the Game of Life on a square grid with periodic boundary conditions. The programme is written in C++, and makes use of optimisation techniques. The code is also parallelised, using MPI and openMP.

1.1 Protoyping

Before starting the actual code, I decided to base my method on what I knew of *GRChombo*, a code I use in my research group for the evolution of gravitational systems. I wanted to start by creating a program that would work on a single processor, though keeping in mind that I would parallelise it later. I knew I wanted to split the grid into subgrids, and have each processor work on a subgrid – this is similar to the box decomposition used in *GRChombo*. I also knew that I would have to communicate the ghost cells between processors, and that I would have to use MPI for this. This lead me to define the `Grid` class, which would contain the entire grid, and the `Board` class, which would contain the subgrid that each processor would work on. I knew that using a separate treatment for the ghost corners would require additional communication between the processors, and therefore I decided to include the ghost corners in the ghost rows.

2 Algorithm

This section describes the overall algorithm and flow of the program, whereas the next section aims to delve deeper into the implementation and optimisation methods. The programme takes as input a parameter file, which defines the following parameters:

- The size of the (square) grid.
- The number of evolution timesteps.
- The interval at which the grid is saved.
- The number of OpenMP threads.
- The option to generate random initial data or to read an existing grid.
- The fraction of live cells in the case of random initialization.
- *(Extension):* The critical number of neighbours determining the rules of the Game of Life. This is 3 in the standard version of the game.

After initialization, the code updates the grid for the demanded number of iterations, saving output when specified. The output takes the form `step*.txt`, with each file containing a (whitespace-separated) binary grid. These text files can then be converted to PNG images using a post-processing tool, for which an example is included in the directory[1].

3 Implementation

I have two different main functions, the second being the parallelised version of the other. I start by describing the basic version, and then discuss the ways I used to parallelise everything.

3.1 Single-thread program

Structure

I have created four classes:

- `GameParams`: contains the parameters as specified in the input, and has methods for reading and displaying them.
- `Array1D`: this is copied from what we have seen in the course. A good way to store 1-dimensional arrays in `C++`. I have added methods to overwrite them, and return sub-arrays.
- `Grid`: this class is based on the `Array2D` class we have seen in the course. I have adapted it to satisfy the needs for this project. Its purpose is to serve as the class containing the entire binary grid. The class contains methods to read and write to parts of the grid, as well as save it.
- `Board`: this class inherits from `Grid`. This class adds all the methods to update the grid, and contains member variables representing the ghost cells.

For the purpose of the single-thread program, there is no difference between the latter two classes in the sense that the entire grid is stored in an instance of the `Board` class. Additionally, there are five functions in `Functions.hpp/cpp`. The first three are for initializing at random, from a file and a function that contains the iteration procedure. The latter two are part of an attempt to make my code more flexible towards the number of MPI ranks, and are discussed in the next section.

The code starts by reading the parameter file, and initializing the grid. Given the periodic boundary conditions, the ghost cells are filled based on the outer rows and columns of the board. This completes initialization, and the update procedure is then carried out for the required number of steps.

The update algorithm consists of the following steps:

[1] This is however a very basic, non-optimized Python routine.

1. The board is updated row per row: for row i, the number of horizontal neighbours is determined for row $i-1$, i and $i+1$, after which these three rows [2] are added together to determine the total number of neighbours.
2. The ghost cells are now updated (based on the periodic boundary conditions).
3. If the board needs to be saved, this is done. The board is now ready for the next iteration.

Optimisation

Given that we have to optimise our code, I decided to use the contiguous array class that we have seen in the course, and base my `Grid` class on the 2D version. I update over the rows, as they are contiguous in memory.

I ran into several errors with memory usage, that I managed to resolve through the use of `valgrind`. Concretely, this led me to introduce the `copy_into` member function instead of `overwrite`, which ended up curing the problem[3]. The latter takes an array, whereas the former takes a pointer to an array. The `copy_into` function is used on the storage arrays in the update procedure.

I have avoided branching in my update routine by finding an explicit expression for the updated value of a cell:

```
> val = data[j];
> data[j] = (1 - val) * (N_nb == N_nb_crit) +
>           val * (N_nb == N_nb_crit || N_nb == N_nb_crit - 1);
```

where N_nb_crit is the critical number of neighbours set from the parameter file. Also, by counting the neighbours horizontally, I have aimed to work contiguously. Once the neighbour row is no longer needed, it gets overwritten by another one.

Results

I have verified the correctness of the code by recreating some of the examples that can be found on Wikipedia, like the small spaceship and the Gosper glider gun. These can be found in the repository, in the `examples` directory.

The code is very fast for small grids; for example, $\mathcal{O}(10^{-2}\,\mathrm{s})$ for 200 iterations of the Gosper glider gun. The time increases as $\mathcal{O}(N^2)$ upon increasing the dimension N of the grid, as expected. This is illustrated in Figure 1, where one can see that the slope of the single-thread curve is the same as the reference line. This figure also shows that the single-thread code takes a couple seconds for a 1000 by 1000 grid, that is evolved for a thousand steps (and saved every iteration).

3.2 Parallelised programme

I have then taken the above single-thread programme, and parallelised it in the following ways.

Now the motherboard is initialised as an instance of the `Grid` class. I have then used MPI to send this motherboard to a collection of ranks, using a two-dimensional Cartesian communicator. Using domain decomposition, I then divided the motherboard into subgrids that align with the coordinates induced by the Cartesian communicator. Every rank initializes an instance of `Board`, on which it only copies the part of the grid it is responsible for. Before the first iteration, the ghost cells can be derived from the motherboard, but once the updating procedure starts the ghost cells must be communicated after every iteration. This is neatly done using the topology of the communicator: first the ghost columns are communicated *horizontally*, after which the ghost rows are communicated *vertically*[4]. Every time the grid needs to be saved, all ranks communicate their part of the grid to rank 0, which then updates the motherboard and saves it.

[2] The board class contains 3 storage arrays, with length equal to the number of columns in the grid.
[3] I must admit that the exact reason this fixed it is still somewhat of a mystery, however …
[4] This is because in my implementation the ghost rows include the ghost corners.

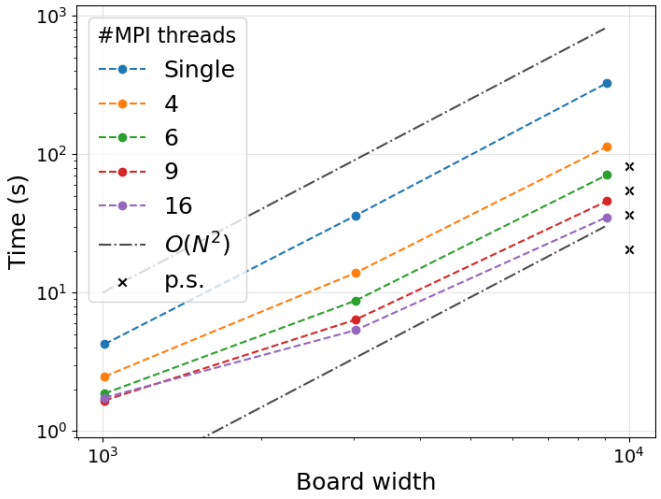

Figure 1: Performance of the parallel code for different number of MPI ranks, compared to the single-core code. The grid is initialised randomly

I have used openMP to parallelise some of the for loops that arise when overwriting and storing stuff. This parallelisation of a for loop also arises when updating a single row in the grid. Note however that the updating procedure still goes one row at a time for every MPI rank.

Results

Some results for the parallel version of the code are shown in Figure 1. These show the speed-up compared to the single-thread version, for different numbers of MPI ranks (but only one openMP thread per rank). We see that, initially, the slope of the increase in duration with increasing board size is less steep compared to that of the single-thread version. In this regime, there is still a significant cost of communication, a contribution that becomes negligible as the board size increases. In principle, the parallel version should be faster than the single-thread version by a factor equal to the number of ranks: this is not the case for small grids, but is approached when the grid size becomes large enough: this is shown by the crosses on the right of the figure, which indicate the duration for the largest grid (right-most dots) if the scaling would have been perfect.

I have noticed that adding openMP ranks actually slows down the code (at least for gridsizes comparable to those in Figure 1) – which is of course not what I want. I believe this is due to the little gain in speed (as updating the rows horizontally is already quite efficient, and so the added communication of the openMP outweighs the gain in speed). Therefore, I have attempted to openMP-parallelise over the rows themselves, by having each thread update a range of rows. This is not yet finalised however, and I have run out of time[5]. A future version of my code would ideally include this parallelisation over the rows, rather than the columns.

3.3 Documentation, style and good practice

I have made sure the code is compliant with the Google style guide through the use of `clang-format`[6], and have generated documentation using `Doxygen`. I have automated these tasks with the use of `Make` and `CMake`. I have also included a couple of unit tests[7], which can be run using `ctest`.

[5] My current attempt is in the `dev` branch, but still has memory errors.
[6] This does not work on CSD3 though, and I have therefore commented it in `CMakeLists.txt`
[7] Though, admittedly, towards the end of the project

4 Conclusion

I have devised a single-thread programme that runs the Game of Life on a square grid with periodic boundary conditions. Doing so, I have implemented some of the optimisation techniques we have seen in class, like the avoiding of branching. This code runs efficiently, and I have verified its correctness by recreating some of the examples found on Wikipedia.

I have then parallelised this code using MPI and openMP, using a Cartesian communicator. This code also produces correct results, but is less efficient than the single-thread code for small grids. This can be attributed to the cost of communication being higher than the cost of calculation for small grids. For larger grids, the parallelised code outperforms the single-thread code – at least when parallelized with MPI. The parallelisation with openMP does not provide a speed-up yet for the cases I have tested. However, for very large grids even the openMP parallelisation might start giving better results.

However, ideally I would like to parallelise the update routine over the rows itself with openMP, as I expect that to provide an additional speedup. I haven't finished this in time, though.

Finally, I have ran a couple examples changing the number of critical neighbours, and I found that most of these simulations died out quickly. Therefore, it seems like the default value of three does provide the most interesting evolution.

Appendix: Use of ChatGPT

I did make use of chatGPT, but only to ask specific questions on how to do something (similar to looking something up on Stack Overflow), and mostly on I/O handling. Specifically, I used it to help me create

- the function to read the parameter file,
- generate random binary numbers along a Bernoulli distribution,
- write to a txt file,
- figure out how destructors work,
- read binary tables
- and finally how to do command line argument parsing in `C++`.

Furthermore I obviously also made use of StackOverflow. I have not made use of ChatGPT to write larger chunks of code, or help me construct the body of my code.

At the end of the project, I have been told that Copilot is free to use for students. By the time I got it, most of the project was finished, but I have used it to document my code. I have only made minor modifications to the code with it.