PROJECT REPORT

FOUNDATIONS OF HIGH PERFORMANCE COMPUTING 2022-2023

Guido Cera

Table of Contents

0. Introduction

The project consists of a scaling study on a personal and highly parallel implementation of the so called Conway's Game of Life.

0.1 Game

Game of Life is played on a grid, where each cell can be either dead or alive. Starting with a grid in a state - with some cells alive and some dead - the grid "evolves", meaning each cell keeps or changes its state depending on the state of the neighbouring cells. The specific implementation for the evolution rules used in this project is presented in the assignment description (github.com) and varies slightly from the classical rules of Conway's Game of Life. The grid is virtually infinite, the outer cells of the grid use the cells at the opposite site as neighbours; imagine that the grid multiplicates and each copy sits adjacent to the original in the same orientation. In a computer the cells have to be updated one by one and this offers multiple possibilities in how the cells are navigated and when they are updated; the two methods I implemented are called **ordered evolution** and **static evolution**. The former is preaphs the simplest both conceptually and in code, it is a loop over all cells in order and each is updated right away, before passing on to the next cell. The latter method holds the update of the cells after all cells have been checked and their next state have been decided. While in the ordered case the neighbours of a cell will be half in the old state and half in the new one, for the static evolution all neighours will be in the same state at the time of determining the new state of a cell.

Interesting fact, the classic implementation of Conway's Game of Life is Turin complete. (Wikipedia)

0.2 Orfeo

This project is being developed to run specifically on the high performance computer Orfeo located in Area Science Park, Trieste (areasciencepark.it). This is important for the parallelization of the code, which has to be built using the resources available on Orfeo (ofeo-doc.areasciencepark.it). Thus the code will contain an hybridization of MPI (open-mpi.org) - for multiprocessing - and OpenMP (openmp.org) - for multithreading.

1. Methodology

From an abstract point of view the core of the project - the Game of Life - is pretty straighfoward since it comprises of a matrix, some nested for loops and few simple checks to update the cells. I approached the project implementing this core structure first and then dwelving in the parallelization; and even if I have experience in multi-processing and multi-threading it felt like going from swimming in a pool to being thrown in the ocean during a storm. The most interesting choice of algorithm I encoutered at this stage was approacing the static evolution [1.1]. Now, I do not want to give the impression that no optimization can be done, on the contrary, there are a great number of articles and book's chapters that analize the problem and make use of clever details of computer architecture or theory on sparce matrices and data structures in general to shrink dramatically the number of operations needed to play the game; here are some links to provide an overview of the optimization landscape for Game of Life.

- (stackoverflow.com) Post with some good answers and links to useful sources;
- (docplayer.net) Make use of theory on **sparce matrices**;
- (jaregoy.com) Michael Abrash’s Graphics Programming Black Book, chapter 17 spends 33 pages to optimize Game of Life;
- (Wikipedia) **Hashlife** is an algorithm that makes use of hash tables and reaches impressive speeds, with the drawback of occuping large amounts of memory
- (dotat.at) An implementation of Game of Life which uses **bit-wise opterations** and **lookup tables**.

I want to stress the many possibilities of optimization since the philosopy of coding for an high performance computer is all about that, but one of the focuses of this course is parallelization, in particular through MPI and OpenMP, and the objective of this project is to show the scalability of multithreading and multiprocessing. In light of all this I decided to focus on implementing a good parallelization and I left a simple representation of the game underneath; which might actually help showing the power of multiprocessing when scaling.

The following subsections explore the coiches made in specific situations or show the methodological approach to specific subproblems.

1.1 Static Evolution

For this approach to evolution I need first to check all the cells and determine their next state and then, as a separate step, I can update the grid. This way the current state of the table is the only one used to update it. The problem resides in the computed next state, where do I put it while I check the following cells if I cannot use the grid itself? I need a way to store the next state until all cells have been checked and their next state

computed. The way I store the next state needs also to be easily accessible and fast to read because as soon as I am done writing all next states there I will read them and transfer them to the actual grid.
I saw two possible solutions to this problem, each with their own positives and negatives; one being to use middle-states to mark cells to modify and the other being to store two separate lists with the coordinates of cells to kill and to reviv. Follows an explanation and then an analisys of the two approaches in terms of memory needed, computational load and ease of use.

- **Middle-states.** With middle-states I mean adding states that a cell can be in other than "alive" and "dead". In particular I add the states "will die" to mark a cell that is alive but will have to die at the next evolution step, and "will live" to mark a cell that is dead but will live at the next evolution step. This approach consists of marking the cells directly on the grid, without any additional data structure, but in a way that still leaves visible the current state of the grid so that checking the following cells will not be influenced by the mark. The drawback of this approach presents itself at the moment of updating the grid for the next state, here I have to change all cells in a middle-state to the state they are meant to have. There are no shortcuts to accomplish this, an additional full scan of the grid is necessary to find the middle-states and write the corresponding final state.
- **Separate Lists.** Two separate dynamic lists store coordinates, one of the cells to revive and the other of the cells to kill. This approach uses additional memory but minimizes the effort to update the grid afterwards, since I only have to loop through the lists and access only the cells that need modifying and no other.

From initial experiments I carried out I formed some expectations on the behaviour of this game. Unfortunately I didn't save the precise numbers but my takeaway is that a significant number of cells change state each evolution step if we start from a random initial state [2.4]. Transalted to the two solutions above, this means that the lists will occupy a significant amount of memory each step, and if I update the memory allocated for each list doubling it when it is full I could have to allocate up to four times the size of the grid at each evolution step. Moreover middle-states are much easier to handle than dynamic memory. All this is why I opted for the middle-states soution instead of using separate lists. Further implementation details can be found in the next section [2.3].

1.2 Ordered Evolution

No OpenMP, it is useless.

1.3 Multithreading

I focused on the heaviest part for multithreading and I left singlethreaded other parts that could have been multi threaded to have a cleaner code. Results back me up in this decision.

1.4 Multiprocessing

- handling of the board (splitting, reconeccting to save etc.)
- all single-process operations handled by root process. This enables single-process execution of the code.
- not bundling together the sends and receives of the propagation for clarity

1.5 Board generation

`random_board()`, talk probability `easy_seed()` I have no requirement of true randomness so I don't care to use the time or whatever fancy way to get a completely different seed every time.

1.6 Read and write image

not implemented by me so no notes on approach or implementation, I just learned how to use them. Same for `swap_image()`.

1.7 Time tracking

multiple times for each run, what they mean, why.

2. Implementation

2.1 Code management

How is the code managed, what is in which files, why is it.

2.2 Cells states

The matrix that stores grid, and also all auxiliary variables and arrays that store cells, are of type `unsigned char`. Firstable `char` is the type that occupies the least memory in C at 1 Byte, which is plenty to store the numbers from 0 to 255 that the grid uses. Secondable since, as just noted, the numbers to be stored are all positive using `usigned` provides an hardcoded guard against negative values; this way the code better resembles the logic behind it.

2.3 Static Evolution

128,127: values to mark cells that will change

2.4 Board generation

`random_board()`, multithreading problem with `rand()` and the seed. https://pvs-studio.com/en/blog/posts/0012/

2.5 `check_neighbours`

`check_neighbours()`

3. Results & Discussion

4. Conclusions