Pierre Osselin
pierre.osselin@eng.ox.ac.uk

Assignment Report

Embedded Systems
December 24, 2020

Abstract

Cellular Automata regroup a class of discrete models of computation that have been widely studied in the 20th century and that found applications in different areas including physics, theoretical biology and microstructure modeling. In this work we implement a graphical interface for a particular cellular automaton on a 2d grid called "Conway's Game of Life" in C++ with the library SDL2. We then explore the possibility for cache optimisation and parallelization of our implementation, resulting in an increase in performance if the grid is large enough.

1 Conway's Game of Life

Definition. A cellular automaton is a collection of cells on a grid of specified shape that evolves through a number of discrete time steps according to a set of rules based on the states of neighboring cells. The rules are then applied iteratively for as many time steps as desired. Cellular automata were studied as a possible avenue of modelisation, particularly for biological and physical systems [3] and have been used in acoustic [2] and cryptography [1].

Update Rule. The universe of the Game of Life is an infinite, two-dimensional orthogonal grid of square cells, each of which is in one of two possible states, live or dead, (or populated and unpopulated, respectively). Every cell interacts with its eight neighbours, which are the cells that are horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:

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.

Implementation. In our graphical implementation we will use a finite grid of size 20x20, even though the definition of the grid can be easily modified in the definitions.h file. We use the libraries "SDL2" and "SDL_TTF" for the graphic and text elements respectively. Our implementation includes the possibility to define the initial seed freely, as well as saving and loading predefined ones.

Figure 1: Screenshots of the graphical interface

2 Cache Optimisation

There are multiple elements to consider when one wants to optimize c++ code for performance. One of them is to optimize the code structure as to avoid unecessary computation. Another one is to favor function inline as to avoid function calling overhead. One element that we will explore here is cache optimisation : when modern CPUs load data from main memory into processor cache, they fetch more than a single value, instead they fetch a block of memory containing the requested data and adjacent data. This can important for us as we iteratively update a 2d array, hence we want our data structure to optimally match our cellular automaton update rules.

Experiment. To highlight the possible increase in performance due to cache optimisation, we used a 1d array of indexing that will be used as a mapping from the 2d grid to the index of the array that stores the value of the grid. We could have used a function mapping but we wanted to equalize the computational complexity of this mapping across different indexing choices. We then perform a variable number of iterations on a grid of 600x600 cells. We consider the following mappings $f : [N]^2 \to [N^2]$ (we suppose $N = 3K$ for $K \in \mathbb{N}$):

$$f_1(x, y) = \sigma(N \times x + y) \text{ for one } \sigma \in S_{N^2} \text{ the set of permutations}$$

$$f_2(x, y) = N \times x + y \text{ (classic grid[N][N] implementation)}$$

$$f_3(x, y) = (x/3) \times (3 \times N) + 9 \times (y/3) + 3 \times (x\%3) + (y\%3)$$

f_1, f_2 and f_3 represent respectively a "random" indexing, a "naive" indexing where the 2d grid is represented like a matrix, and the "optimized" indexing, where the indexing is by 3x3 squares. Since the update occur is carried out cell by cell in the reading order, the "naive" approach is likely to perform 3 cache importations (for large grids) compared to 2 maximum in the optimized case (the worst case is a corner in the indexing which involves two pairs of two consecutive blocks).

Result. We perform our experiment with $N = 600$ and run the process from 0 to 300 iterations. We plotted with cpp gnuplot shown in figure 2. We don't perceive a significant different between the Naive and Optimized approach, which suggests that the computational bottlenecks in our implementation do not come from the memory access. Compared to the random approach, we can see a roughly 10 percent improvement in computational performance.

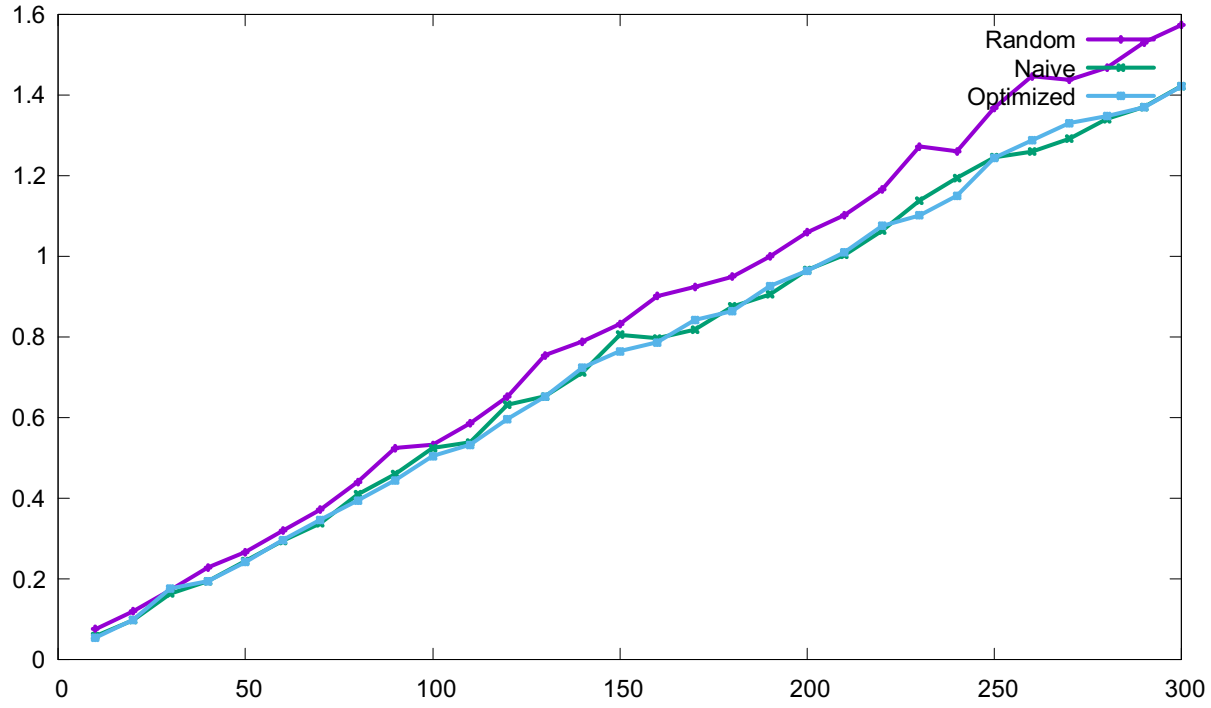

Figure 2: Running time in seconds by number of iterations for different indexing strategy

3 Parallelization

Cellular automata consist in updating a grid according to an update rule in an iterative fashion. Another method that might increase performance is parallelization of the grid update. The parallelization is only possible if the grid is separable with respect to the update mapping. In our case, the update occurs locally (every cell is updated via its

neighbors), hence this method is applicable. In this section we will explore the potential to parallelize our implementation.

Experiment. In this experiment we will divide our grid in four "augmented" quadrants (if we have a $(2N+1) \times (2N+1)$ grid, we consider the four $(N+1) \times (N+1)$ quadrants (their intersection is not null)). In every iteration we create four threads which aim at performing an updating on every one of these four quadrants. In addition, we also perform an update without any threads on another full array directly. We perform a variable number of iterations of this update for different grid sizes. We carry out this experiment for 50, 100 and 150 quadrant sizes (or (98, 198 and 298) grid sizes).

Results. We plotted the result of our experiment in figure 3. We can observe that, for a small grid size, multi-threading decreases computational performances. For a large grid size, the performance is significantly increased with multi-threading. This is due to the additional computation involved for performing multi-threading (creating threads, joining threads ect...) compared to updating an array directly.

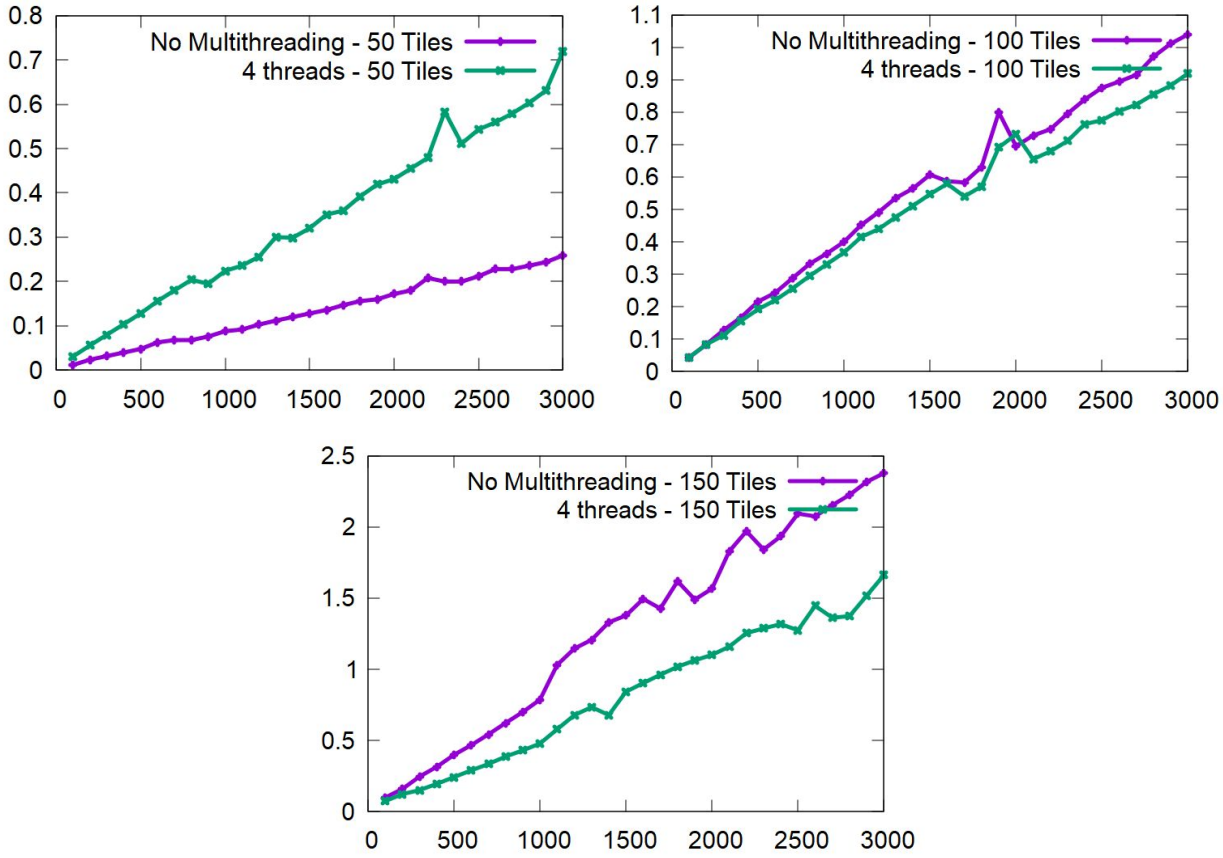

Figure 3: Running time by number of iterations for different size of quadrants

References

[1] Stephen Wolfram. "Cryptography with cellular automata". In: *Conference on the Theory and Application of Cryptographic Techniques*. Springer. 1985, pp. 429–432.

[2] Daniel H Rothman. "Modeling seismic P-waves with cellular automata". In: *Geophysical Research Letters* 14.1 (1987), pp. 17–20.

[3] B Chopard and M Droz. *Cellular automata*. Vol. 1. Springer, 1998.