Game Of Life

Lowe Lundin

June 2019

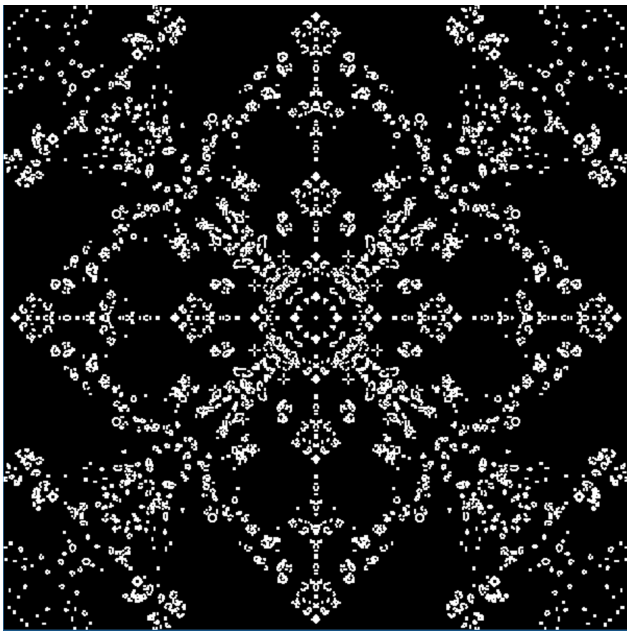

Figure 1: Pattern generated from a single square on the perimeter as initial condition.

1 Introduction

In the course Parallel and Distributed Programming, the last task was to do an individual project, for this "Game of Life" was chosen which is a cellular automata, where every cell on a grid is affected by those around it, meaning a large number of computations is needed.

2 Problem Description

Game of life is a simulation in which special rules and initial conditions completely determine the outcome. The playing field is a grid of any size and every cell can be either "dead" or "alive". Each timestep, the cells change depending on the number of living cells they have adjacent. The rules are:

1. A new cell is born if it has exactly three neighbours
2. A cell dies if it has more than three neighbours
3. A cell dies if it has less than two neighbours
4. For all others cases, the cells remain unchanged

Different initial conditions will therefore decide the outcome of every future timestep.

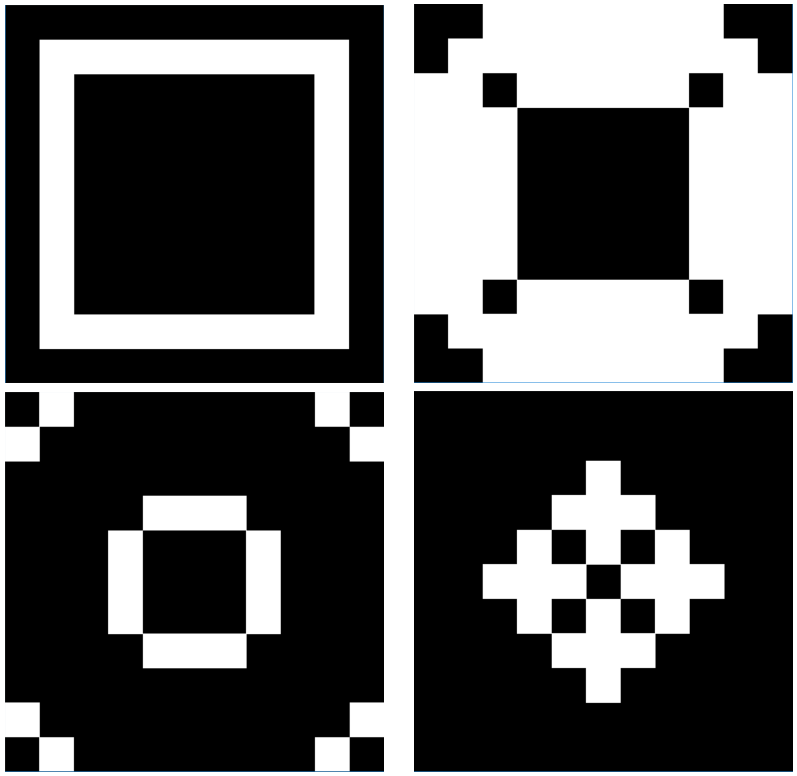

Figure 2: How three timesteps with the given rules looks on a 11x11 grid (white and black cells are living and dead respectively).

3 Solution Method

Since one step of the function only needs the information given from the previous boardstate, the problem is nearly perfectly parallelisable. The board is first split into equally sized collections of rows, where:

$$rows = \frac{boardlength}{processes} + 2 \tag{1}$$

The two extra rows are added for the processes to be able to calculate the next step for the cells on the top and bottom row. After the split, the computation of the next step can then be carried out on the different cores independently.

3.1 Zero-Padding

Zero-padding is a method used in for example Convolutional neural networks, where zeroes are added to "encapsulate a matrix". An additional zero is added at the start and end of every row, as well as two rows full of zeroes at the start and end of the matrix. It is used in this optimisation to be able to remove checks that otherwise have to be done on every node, to make sure that the node for which neighbours are currently computed is not on the perimeter, which would result in an out-of-bounds read on an array.

3.2 Parallellisation

There are two types of parallellisation in the code. One is for when the graphics is enabled and the other one is for when it is off.

3.2.1 Without graphics

When graphics is turned off, the starting array is first scattered into the different processes, with the number of rows for the whole matrix divided by the number of threads plus two rows at the top and bottom sent to their local arrays. The top and bottom row are added, since Game of Life needs to know the surrounding rows to calculate the value of a cell in the next time step. The processes then calculate the next timestep for the local array. Thereafter the penultimate row at the top and bottom is sent to the process below and above in rank respectively, where it replaces the bottom and top row respectively, so the boundary is correct for the next timestep. This process is then repeated, but without any scatter, as the local arrays are all already updated.

3.2.2 With graphics

When graphics is turned on, the process is at first the same, but after the next timestep has been calculated, the data from each process needs to be gathered in the master process, in which the drawGraphics function is called to update the graphic interface. The loop is then repeated with the gathered matrix once again being scattered to the processes. Note that this type of execution is not as effective as running without graphics, and that running with graphics also involves additional function calls, this "with graphics-mode" was simply included for fun.

4 Experiments

The experiments were carried out on the UPPMAX Rackham cluster, which runs Intel(R) Xeon(R) CPUE5-2630 v4 @ 2.20GHz. The code was originally made so that it read the inputs from a txt-file generated by a separate program. The limited storage on Rackham did however force a solution where the inputs are generated in the code itself. Note that all experiments were made for the solution without graphics and that this solution does not function properly for single core runs, meaning that all speedup is calculated with respect to 2 core runs. The executions were only done for 10 timesteps, as it is not especially interesting to look at different number of timesteps, as opposed to the grid size.

The runs were made on grids with sides of length [128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32786]. It would have been preferable to continue further, but the program crashes due to overflows when $N * N > 2^{31}$. As runs on more than 64 cores would need larger grid sizes to be effective, these were omitted as an effect of previously mentioned restriction and the number of cores run on was [2, 4, 8, 16, 32, 64]. The correctness of the runs were verified with the graphic interface.

5 Results

Grid size	2 cores	4 cores	8 cores	16 cores	32 cores	64 cores
128^2	0.003492	0.002536	0.001213	0.000889	0.239991	0.539950
256^2	0.016121	0.007767	0.004605	0.003013	0.190802	0.580311
512^2	0.058113	0.030628	0.015334	0.008680	0.224236	0.581344
1024^2	0.219140	0.113578	0.054869	0.029995	0.319804	0.881666
2048^2	0.819600	0.433811	0.216370	0.121551	0.336274	0.959869
4096^2	2.631794	1.397462	0.758629	0.446057	0.495057	1.050044
8192^2	10.061356	5.079947	2.837199	1.608260	1.156995	1.323471
16384^2	39.763251	20.217699	11.108308	6.413146	3.545106	2.559369
32768^2	160.586800	81.797124	44.640107	26.328606	13.739139	7.851221

Table 1: Wall time given in [s] with varying number of cores and input size.

Parameters	Time in seconds
$n = 8192^2, p = 2$	10.061356
$n = 16384^2, p = 8$	11.108308
$n = 32786^2, p = 32$	13.739139

Table 2: Weak scalability.

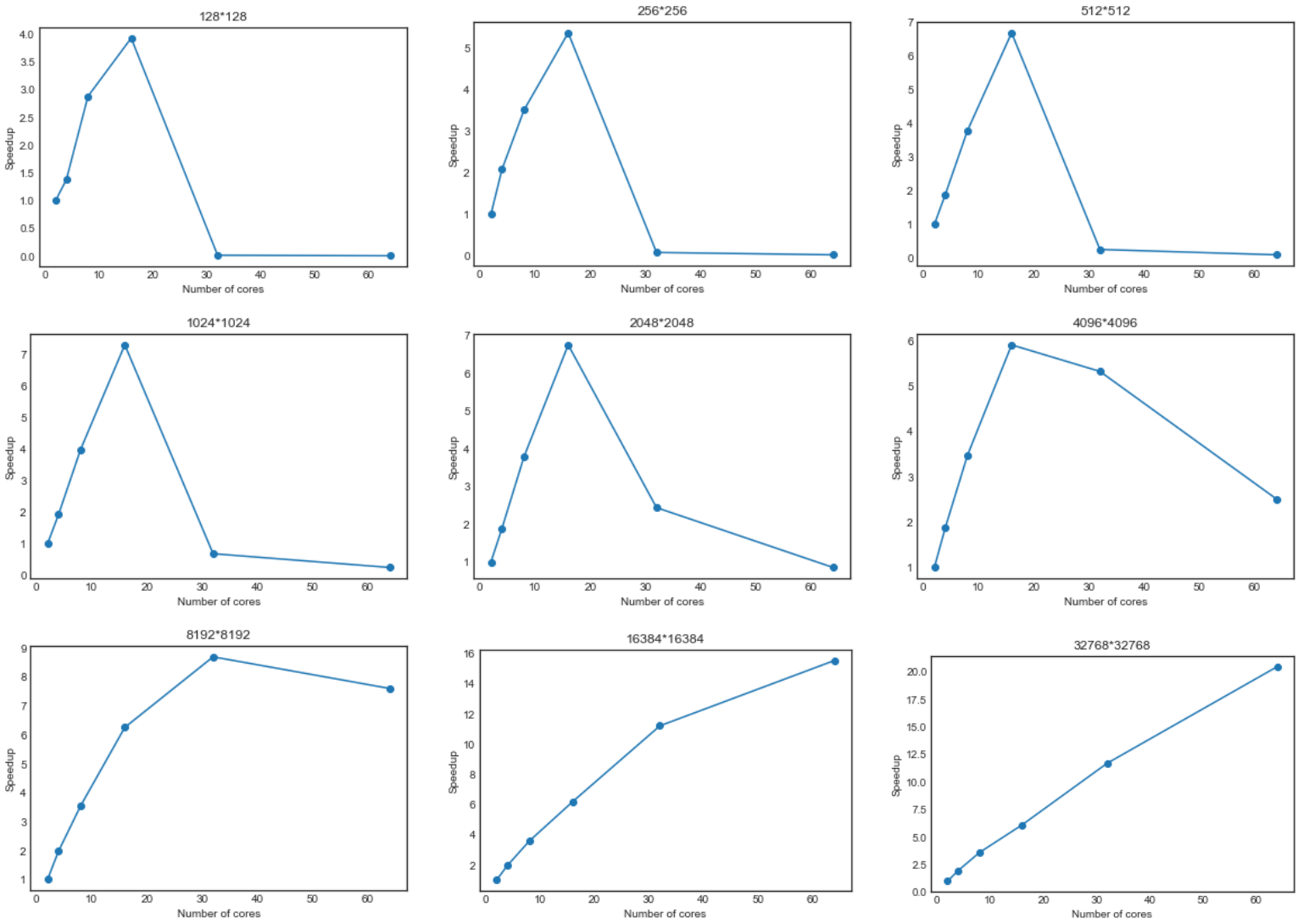

Figure 3: Speedup with increasing number of cores and different grid sizes. Note that the speedup is calculated with respect to the runs on 2 cores.

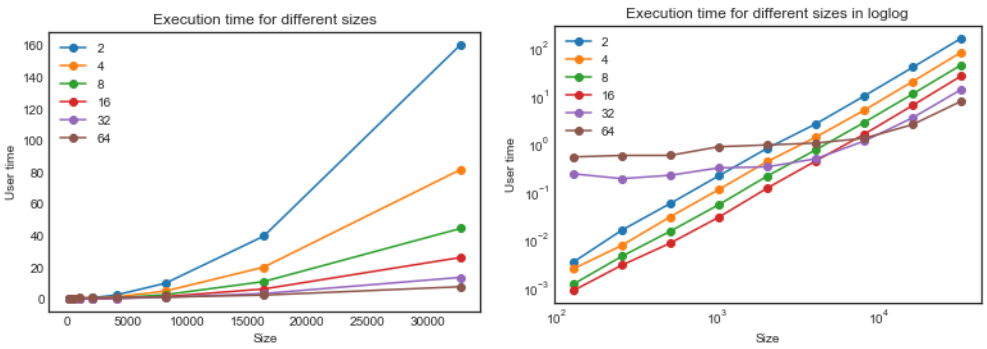

Figure 4: Wall time for different numbers of core and grid sizes, note that "size" is the length of a side, meaning that the grid size is the square of the given number.

6 Conclusions

It can be observed from table 1 and the plot in figure 4 without loglog axises that as the number of elements is multiplied by four, the wall time follows in an almost perfectly linear fashion for the lower number of cores. This is however not the case for 32 and 64 cores, as those are ran on grids which in theory are too small for them to reach their full potential, as communication starts being a problem by taking up a large proportion of the execution time. This is especially evident when looking at the loglog plot in figure 4, where it can be seen that the lines representing the wall time are almost all equidistant, with the exception of the 32 and 64 core runs, which do not perform as well on smaller grid sizes, but do "fall in line" somewhat as the grid sizes gets larger, though not as well as one would hope. Another, perhaps more important, factor for the performance of the 32 and 64 core runs is that the Rackham nodes contain 20 cores each and that the 32 and 64 core runs need to be executed on 2 and 4 nodes respectively with added communication costs as a result.

The problem with generating larger grid sizes also means the weak scalability is not as good as it should theoretically be. As the problem is fairly straight forward with a simple split of the matrix, the theoretical complexity would be $\mathcal{O}(n)$, however communication costs and the problem with not being able to generate large enough grid sizes without the program crashing means the weak scalability is not what it should optimally be.

In figure 3 one can see how the speedup is very reliant on the grid size. For smaller grid sizes, the speedup for a larger amount of cores is well below one. For a smaller amount of cores, the speedup is very good, for example for the 1024*1024 run, where the 16 core run reaches above 7, not too far away from the theoretical optimum of 8. The speedup for 32 and 64 cores is better than the one for 16 cores after the 8192*8192 run, however it is expected that these runs would be able to achieve even better speedups given larger grid sizes.