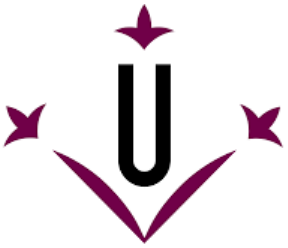

Universitat de Lleida

Game Of Life parallelization with OpenMP

Made by
Oriol Alàs Cercós, Sergi Simón Balcells

Delivery
17th of March, 2022

Universitat de Lleida

Escola Politècnica Superior

Màster en Enginyeria Informàtica

High Performance Computing

Professorate:

Francesc Giné

Jordi Ricard Onrubia

Contents

List of Tables

List of Figures

1 Introduction

In this project, it will be discussed how we have parallelized the code of the game of life provided by the professors. First, we will talk about the design choices that were done during the development, as well as some changes that were made to the sequential code to make the analysis of speed up and efficiency more true to the game of life.

Then, we will provide the results of this code executed at the Moore cluster with the examples of 5000 and 10000 matrices provided in the assignment. In the same section, we will analyze why the application behaves like that.

Finally, a conclusion to the analysis is provided, where we remark on the key points that were discovered in the analysis.

2 Design Choices

There are 6 loops parallelized on the modified code that was given by the professors. The first two that could appear in the code are for generating randomly the grid with the given parameters. It may be weird to see it is only this kind of loop that is parallelized, but reading data from a sequential disk concurrently only makes it gain little to no speedup and brings down the efficiency of the data is not processed extensively. In our case, as it is a simple assignment to our grid variable, it only makes it perform worse.

The next two loops correspond to the update of the grid for each step. The first is for computing a second grid that counts how many cells are alive. This makes the second be able to compute easily and faster if the cell will be dead or alive. The last two loops correspond when a grid is read from a file, as there are two loops in this step to set the borders of the grid to dead.

Moreover, There were some changes made to the serial code that was provided by the professor. The code wrote the steps for checking the application in the main loop. As said before, writing in a file system is costly as it can only be done sequentially as well as is slower than the CPU capacity. For this reason, we made the steps stored for debugging an optional activated by the constant debug. This was made solely to study better the

speedup and efficiency of the program.

There was a decision that was left to be empirically demonstrated between different executions, which was which type of schedule should the application use. We tried to do it dynamically, statically, and with guiding policies. The dynamically and statically performed badly in comparison to the guided one, with the statically even worsening with the number of cores to parallelize. Nevertheless, the guided strategy performed better than all of them, and, as we will see in the analysis, the efficiency and the speed-up with different cores are almost linear.

In addition, the code was compiled with `-O3` flag from the GCC to activate the optimizations. It performs an order of magnitude better than the code not optimized. This is not surprising as the loop contains an if-else that sets the boolean condition for the cell to be alive or dead. With the `branchless` optimization, most of the statements inside a loop are removed in favor of multiplication to set the variable inside them. This means that the hardware can compute more efficiently the instructions, and, as said, Game of Life makes extensive use of this optimization.

Finally, was added a parameter to the execution. The full execution is:

```
lifegame cores [name-of-file] [x-size] [y-size] [max-iterations]
```

This lets us make it easier to execute the application in the backend without needing to recompile it each time.

3 Results

The results are gathered in the table 1 2, 3. There are different graphics that can be seen in these application, resuming the fact of how evolves the time, the speed-up, and the efficiency in the figures 1, 2, 3, 4 respectively.

The time measured in these graphics and tables is the time that the application needs to read the file and update until the maximum number of iterations is met.

Table 1: Time table. There are two examples used in this process, both being square matrixes of 5000 and 10000 specifically.

cores	5000	10000
1	1084.06	8570.89
2	564.25	4449.11
3	381.64	3015.89
4	301.34	2366.07

Table 2: Speedup table. There are two examples used in this process, both being square matrixes of 5000 and 10000 specifically.

cores	10000	5000
1	1.00	1.00
2	1.93	1.92
3	2.84	2.84
4	3.62	3.60

Table 3: Efficiency table. There are two examples used in this process, both being square matrixes of 5000 and 10000 specifically.

cores	10000	5000
1	1.00	1.00
2	0.96	0.96
3	0.95	0.95
4	0.91	0.90

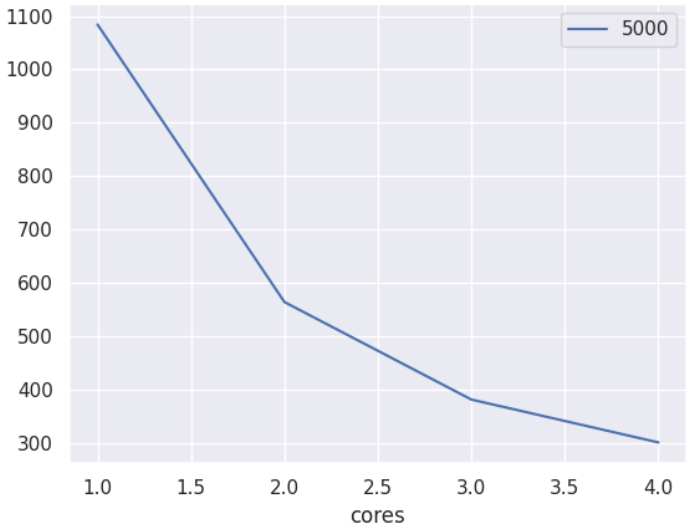

Figure 1: Time plot. The y axis is the time in seconds while the x axis is the number of cores used. This is only for the 5000 sized matrix.

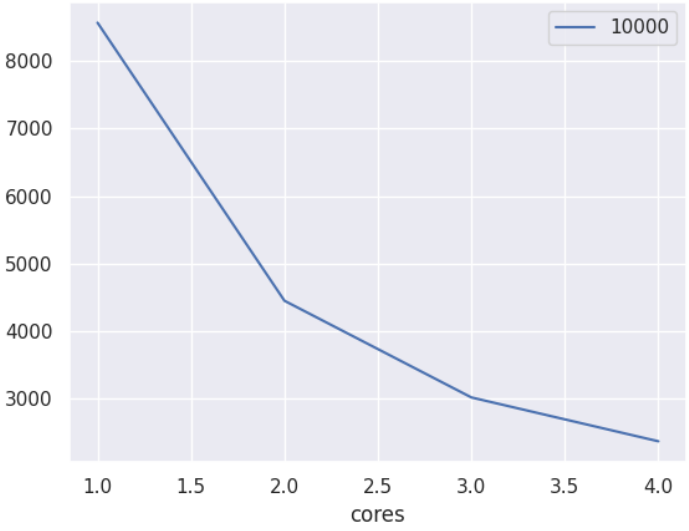

Figure 2: Time plot. The y axis is the time in seconds while the x axis is the number of cores used. It contains the data from the 10000 sized matrix.

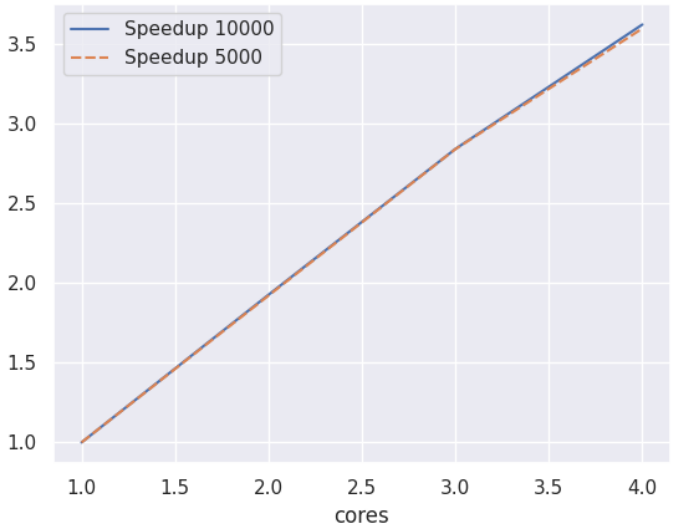

Figure 3: Speedup plot. The y axis is the speedup while the x axis is the number of cores used.

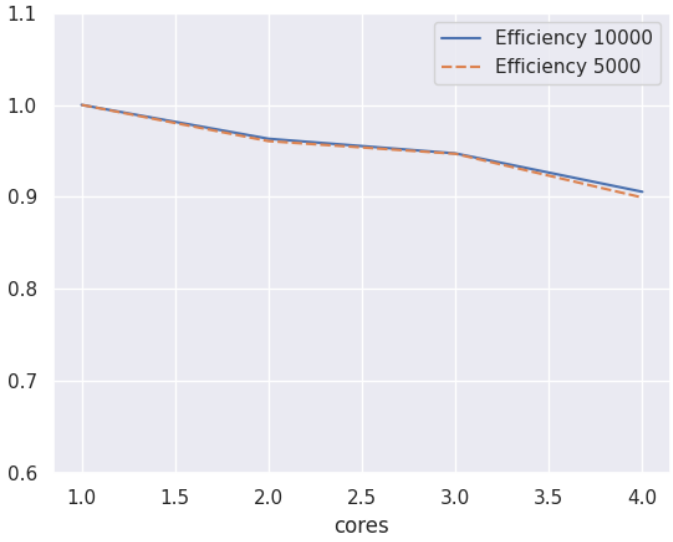

Figure 4: Efficiency plot. The y axis is the efficiency while the x axis is the number of cores used.

3.1 Analysis

This application is almost embarrassingly parallel, as the costs and the speedup increase linearly with the number of cores that uses them. But, there is a small bottleneck that is reading the file: as the disc storage is not concurrent, the reading must be set to read in parallel. For this reason, it can be seen that the efficiency worsens in each core added, as

that part of the program is sequential.

This pattern can be seen more clearly at the efficiency plot, as the efficiency limits to the part of the code that can not be done concurrently, as shown in the next formula:

$$\lim_{x\to\infty} \frac{T_s}{(T_{\text{read file}} + \frac{T_p}{x}) \cdot x} = lim_{x\to\infty} \frac{T_s}{T_{\text{read file}} \cdot x + T_p} = 0$$

This means that the efficiency will eventually reach 0, as the time spend reading the file will never be parallelized. This also means that the maximum speed up is:

$$\lim_{x\to\infty} \frac{T_s}{T_{\text{read file}} + T_p} = \frac{T_s}{T_{\text{read file}}}$$

What's more, this pattern is shockingly the same with both benchmarks that were tried, where at two decimals precision, only with four cores there is a change between them in the efficiency. The aggregated difference in the speedup is also minimal, adding up to 0.3.

4 Conclusion

In this project we parallelized the algorithm of the Game of Life, reaching almost embarrassingly parallel levels of concurrency in the program with the correct use of the scheduler and the optimizations in the GCC set to maximum

Moreover, it was determined that the reading part of the game of life takes is the only sequential part of the whole operation, which makes the efficiency slow down as the number of cores rises when eventually it will reach 0. Additionally, the speedup is theoretically relative to the time spent reading the file.