LifeGame and OpenMP

Paolo Rosa

May 11, 2022

1 Introduction

In this report, a comparison between serial and parallel code is presented for the *lifegame.c*. The tests have been executed in Moore cluster located in EPS, Lleida; 8 nodes cluster of SMPs.

2 Lifegame

The game of life is a game based on a two-dimensional space filled with a population of organisms (cells) which, on the game board, appear like squares. Each cell has 8 cells around itself on all directions, including the diagonals (these are called neighbours cells). Since it is a zero-player, its evolution is determined by the initial state and does not require any subsequent data input. Cells have two states: they can either be dead or alive. The state of a cell change during discrete units of time and it is updated with the status of all the other cells present on the game board. There are four rules:

- *C*ell without life: if the cell has less than two living neighbors
- *L*ive cell: if the cell has exactly two living neighbors
- *C*ell is born: if a cell has exactly three living neighbors
- *D*eath: if a living cell has more than three living neighbors

The objective of our parallel application is to simulate the evolution of the described system during a period of time.

3 Serial code

3.1 Computational cost

Before paralleling a code, it is important to understand the main parts of a program. In this way, we can find the functions that require the most computational cost so that they can be improved thanks to parallelization. The program "lifegame.c" works thanks to three main functions:

- *life_init()* it takes care of allocating the right space for the matrix "grid" and filling up according to the provided inputs
- *life_update()* it takes care of collecting data for each "grid" element from the neighbourhood ones. Then it uploads the grid matrix according to the "lifegame" rules.
- *life_write()* It takes care of writing the final updated iteration on a txt file; and any other requested iteration.

It is possible to measure the computational cost by applying properly the *omp_get_wtime()* function and collecting some data.

In order to measure this difference, the serial code has been executed with 2 different problem sizes (5000, 10000 matrix dimension) under different iterations numbers to show up the contributions of the 3 main functions. The "total" curve is computed within the code and not as sum of three parts; for

consistency and also because in this way it is possible to distinguish better the update contribution function from the total one.

In figure 2 and 1, we can see how the different parts of the code perform for the 5k and 10k matrices.

Figure 1: Execution times of the tree main code parts for 5k matrix

Figure 2: Execution times of the tree main code parts for 10k matrix

As we can see in the graphics, the updating function represents the most important contribution over the total computational cost when the problem size is related to such higher matrix dimension. Since we are interested in dealing with bigger problem, it is useful to parallelize the portion of the *life_update()* function. The *life_init()* and the *life_write* contributions are overlapped in the graphs since they have always low and similar values. Indeed, these functions are called once per time, so their contribution are negligible.

4 Parallelized code

4.1 Life_update function

Since *life_update* has the main computational cost, its core can also be inspected. There are two main for loops: the first one is up to collecting data from the neighbours cells for each box cell in the grid matrix, summing them up in a new value, and saving them in a sum matrix on the same grid matrix position. The second one has to read the sum matrix and update the value on the grid matrix according to the lifegame condition.

4.2 First for loop

We can apply a very simple parallelization clause in the first loop. The directive will impose the creation of a parallel region with a number of threads imposed by the shell script written for the cluster. The cluster is composed by 4-core SMPs, so the number of threads is in a range of 2 - 4.

$\#pragma \quad omp \quad parallel \quad for \quad private(i, i\_prev, i\_next, j\_prev, j\_next)$

Here, we can see the OpenMP directive. Thanks to it, we create a parallel region with default dynvar sets to 1 (so threads = cores = 4). The for loop iterations will be divided equally and associated before the actual execution (static schedule). As private variables, there are the ones that we need for locating the indexes of the neighbour cells and the i variable that is associated with the internal for loop in charge of scrolling the rows. The external j-column index has already privatised by OpenMP for directive.

The first for loop is in charge of collecting data from the neighbour cells for each single value (box) in the initial grid matrix. First, the for loop has to define the proper indexes in order to position the neighbourhood boxes for each i-column and j-row. In this way, the for loop can read and sum up the values and write in a sum matrix of the same dimension of the grid. The i and j are iterated though the matrix to deal with the single value.

By imposing a parallel region, with the proper private variables, the work will be split up among the different threads. Since the code is supposed to work for each random matrix of 0 and 1, the static schedule -imposed by default- seems to be a proper solution to this kind of problem that is load balanced. This statement will be proved in the dynamic schedule chapter.

If the threads can access to the shared grid matrix without constriction there is no need for them to communicate between each other. So there is no time spent for communication between slaves. Instead, the synchronization can be a problem if not proper solved.

But thanks to the omp for directive default settings, we are sure that we have a proper updated sum matrix at the end since there is a omp barrier in the end of the for loop. Unfortunately, it imposes some idle time since, if a thread finishes its work before the others, it needs to wait for the others to start the following instructions.

4.3 Second for-loop

The second for loop will upload the final matrix according to lifegame conditions. So it has to access to the shared grid matrix. The work is different from the first for loop and some thread can work a bit less than the others, but the random distribution may impose the same balanced work. This will be verified in the dynamic schedule chapter 6.

Like the first loop, a synchronization is needed and is guaranteed by the *pragma parallel for* clause. In this way, we are sure the *life_write* will be executed after. There is no need of communication among the threads.

$\#pragma \quad omp \quad parallel \quad for \quad private(i)$ OpenMP directive

Here, we can state the same consideration for the first for loop. We have less private variables to be declared.

5 Comparison

First, a comparison between the serial execution and the parallel one is described in execution time, speedup and efficiency. Then, the dynamic schedule will be applied to understand if there would be some improvement.

The data are collected several times for each iterations number (at least 5) and an overall average is calculated after excluding the outliers. It is important to highlight that there are always some outliers for each execution groups without any preferences for number of processors of number of iterations. Since the initial matrix is always the same as well as the Moore cluster cores, for the presence of outliers we can blame the execution queue: codes with different parameters were executed simultaneously, so the overheads imposed by the bigger ones influenced negatively the execution of the little ones.

The codes are tested for the 5k and 10k matrix dimension. A different number of iterations are provided for the tests. The data are shown in execution time, speedup and efficiency. The execution time is the parameter we used to get numerical data from the code, the speedup is useful to understand

where is the ceiling the parallel version could tend to and finally there is efficiency that let us compare different parallel versions, understanding if increasing the computational resources is worthwhile.

5.1 5k matrix

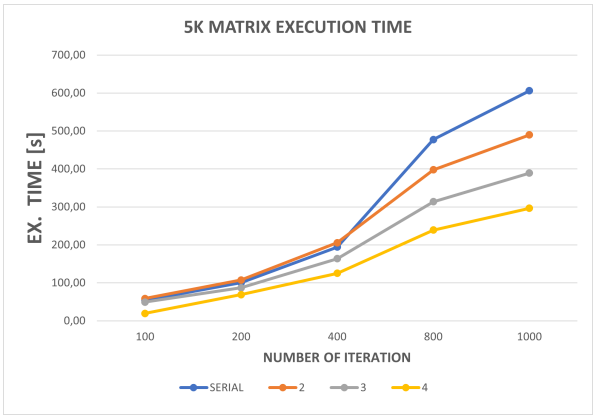

Figure 3: Execution time of the 5k matrix for different processors and iterations number

In figure 3, there is a reasonable order of the execution time curves for the medium-high iterations values. But before the 400 iterations the parallelization seems valuable just upon the 2 processors number. This can be related to the fact that idle time percentage over the total work becomes negligible when when we increase the number of iterations.

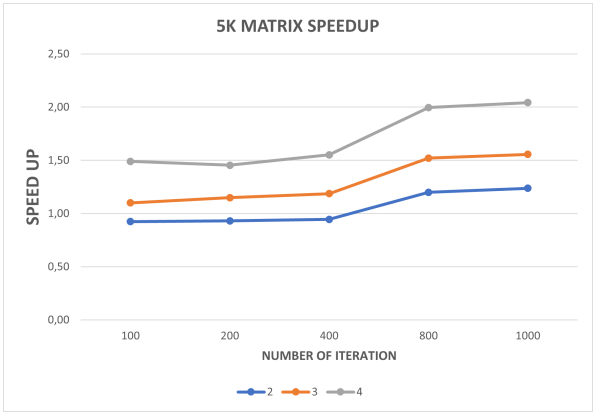

Figure 4: Speedup of the 5k matrix for different processors and iterations number

In figure 4, we can see that for high number of iterations the code seems to perform very well reaching the ideal value of 4. While the 2 -3 processors results seems very similar in speedup. The scalability is quite insufficient since increasing the processor numbers does not seem worthwhile.

Through the efficiency values in figure 5, we can highlight how the 2 processors performs very well -in terms of cost- respect to the others but that the general improvement in efficiency is just important for the 4 processors. The difference can be related to the fact that the idle time is less in 2 processors than in 4 since there are less units that have to wait the slowest.

For the 5k matrix, there are also data for the 10000 iterations size problem, that are shown in chapter 7.1.

5.2 10k matrix

In the execution time chart in 6, the serial code seems to perform as the 2 processors code so there is a difference from the 5k problem size where there is at least a little improvement. The parallelized

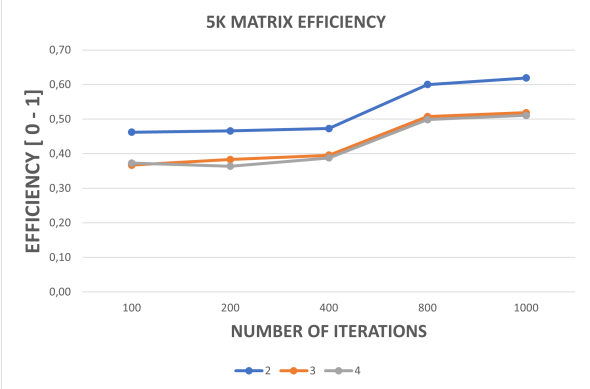

Figure 5: Efficiency of the 5k matrix for different processors and iterations number

versions show up the same curve behaviors but this do not ensure an improvement of performance. Just that the scalability of the problem for the iteration is correct for every number of processors.

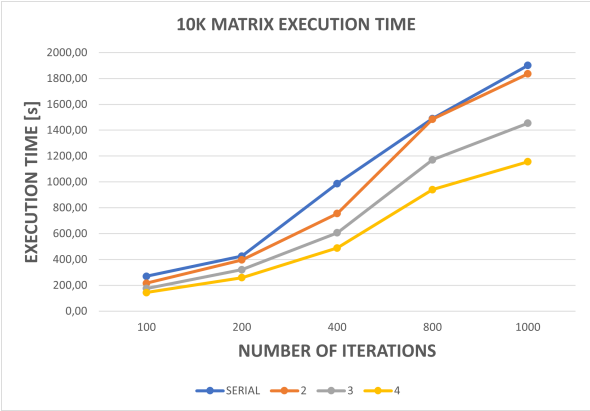

Figure 6: Efficiency of the 10k matrix for different processors and iterations number

The speedup chart in 7 and the efficiency one 8, indeed, show up such little improvement. The speed up

The efficiency seems to prove that the 2 processors parallelization performs very well and that an higher number of processor could not improve the code, increasing cost. But the execution times chart states that the 2 processor's code doesn't seem so worthwhile. So there is a bit of inconsistency in this group of data.

There are no 10k iterations chart for the 10k matrix, since there were no possibilities to collect it. The Moore cluster is limited to 3h in job executions. And if the curves behave linear, an execution time of 5 hours is required for a 2 processors parallel version.

6 Dynamic schedule

Some efforts were spent to analyse how the code behaves in dynamic schedule. For this reason, some data are collected after applying the dynamic clause to the for loops of the *life_update()*. The tests were performed on the 1k grid matrix with a constant number of iterations of 100. The chunk size is defined at the top of the code so it is easy to change it before a new execution. The number of threads is set to 4 for all the executions

```
    #define CHUNK_SIZE 10
#pragma omp parallel for private(i) schedule(dynamic,CHUNK_SIZE)
```

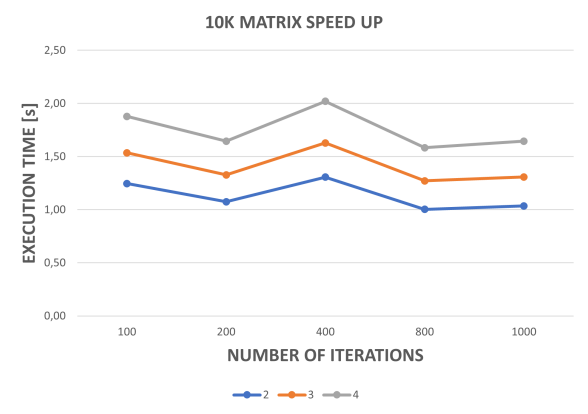

Figure 7: Efficiency of the 10k matrix for different processors and iterations number

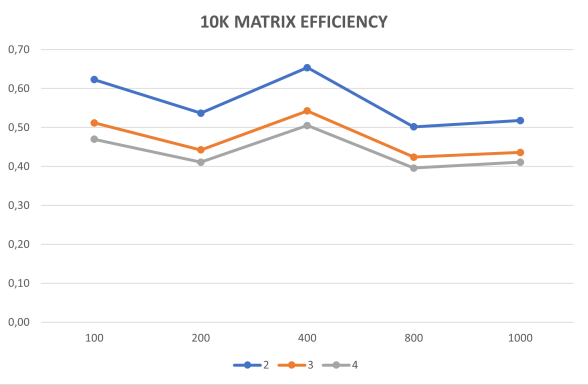

Figure 8: Efficiency of the 10k matrix for different processors and iterations number

Different chunk size were tested in 9. The number of chunk can cover ideally the range 1 - 1k, the maximum grid dimension. But we portioned the for loops in different chunk size of reasonable values since it doesn't make sense to go over 250, the number the static code works with.

The idea was to try to apply the dynamic schedule to the only first for loop (SOLO I), then to the second one (SOLO II) and in the end to both of them (BOTH). The static value are also in the chart in order to highlight eventually the difference and so the improvement.

The result shows that the code seems to work well in dynamic schedule just for the second loop. So, it proves the hyphotesis stated in the section 4.3 that the second loop was not load balanced. So, even though now the gain does not seem worthwhile, since the dynamic values are not smaller than the static one, the code could improve its execution time for bigger number of iterations.

In the end, it is important to notice that the dynamic schedule code showed up some outliers but maybe it is related to the not-deterministic dynamic workflow, so there were not excluded by the average.

7 Other efforts

Some efforts were spent to improve the code but none of them were worthwhile. the first idea was to reduce the idle time associated to the barrier of the first for loop. But merging the 2 for loops was not a solution as explained before, since, in this way, it is not possible to guarantee the correct updating.

To improve the second for loop, the idea was to privatise the variable grid and then include a critical section for uploading the final values. But it is not possible to include the critical clause inside the i and j for loops in charge of scrolling the grid matrix.

A possibility can be implementing some code lines in order to avoid that columns of the same thread waste time recalling the common values from the main memory.

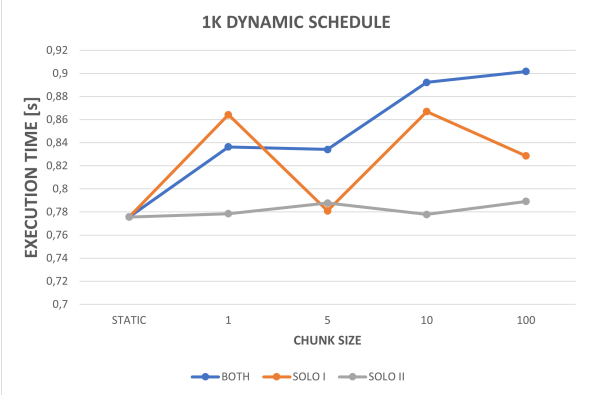

Figure 9: Dynamic schedule effects on different for loops

For the first loop, there is also the possibility to divide the reading from the sum process. It could be worthwhile to implement it through hyper threading (HT), the Intel's version of Simultaneous Multi Threading (SMT). This technique implies the creation of different threads for each physical core, sharing the same functional unit [1]. If we have for example 2 threads inside one core, one of them can access to the shared grid variable while the other one does the sum of them. The main inspiration article for this idea was an article about the effort to find automatically the correct number of thread for each core[2].

7.1 Other charts

In this section, I collected other data from the 5k matrix. The aim was to check if the problem size scaled well.

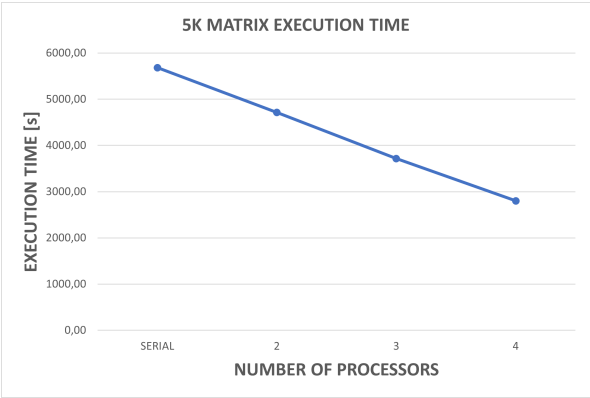

Figure 10: Execution time for 10k iterations problem

The slope of the curve in 10 is quite constant, so the behavior is linear. Unfortunately, it is not so sloped: the contribute of the parallel code is quite good but sufficient. It is possible to evaluate this fact in the speedup chart in 11. For 4 cores, the speedup is far away from the ideal 4 values. Indeed, the efficiency chart in 12 shows up that over the 3 threads it is not worthwhile to increase the number of threads. We could take in account that the parallel clause works only on *life_update*.

7.2 Guided schedule

As the dynamic schedule, a guided clause was also tested. Results are quite similar to the dynamic schedule. For example, for the 10 chunk size, we had average of 0,95815 s, similar to the ones in 9. Since the work is quite well balanced, there is no need to applied a guided schedule. This kind of schedule works well for not balanced work with a clear growing direction.

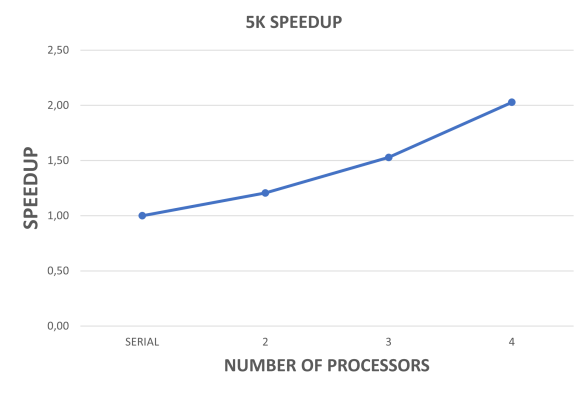

Figure 11: Speedup for 10k iterations problem

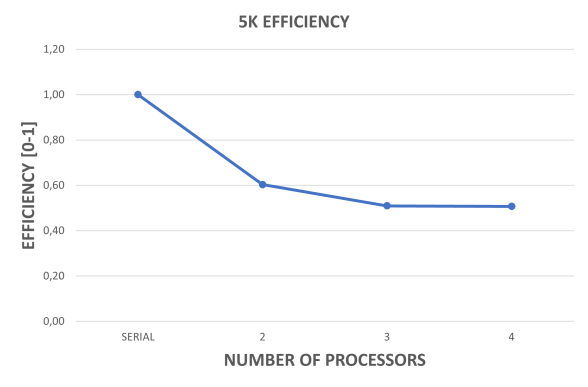

Figure 12: Efficiency for 10k iterations problem

References

[1] Jülich Supercomputing Centre. *Simultaneous Multi-Threading*. URL: https://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/de-installedSystems/JUROPA/UserInfo/SMT.html;jsessionid=20A7517FFC9206B05CBD37F2550A5750?nn=1079852#Start.

[2] Yun Zhang et al. "An Adaptive OpenMP Loop Scheduler for Hyperthreaded SMPs". In: *PDCS*. 2004.