Conway's Game of Life: A High-performance Computing Perspective

Federico Minutoli*
4211286
fede97.minutoli@gmail.com

Matteo Ghirardelli
4147398
matteoghirardelli01@gmail.com

Daniel Surpanu
4120700
surpanudaniel@gmail.com

Abstract

In this project, we use John Conway's Game of Life as a tool to demonstrate and verify the advantages offered by High-performance Computing clusters on a computationally demanding problem. We first introduce some of the theoretical notions behind Game Of Life, as a support for a better understanding of the problem and the implementation of different parallelization models. We then describe the main parallel algorithm design choices, give an exhaustive elicitation of the motivations driving these choices and discuss the implementation logic. In particular, we describe a set of algorithms including a serial program, a shared memory procedure which uses OpenMP, a non-shared memory one using MPI, an hybrid version taking advantage of both OpenMP and MPI, and a GPU-based program using CUDA. In the end, we present the experiments which were launched on the clusters of the University of Genoa to determine the performance of each implementation, perform a critique analysis of the corresponding results, and draw the main conclusions from this project.

1 Introduction

In today's data-driven world, high-performance computing (HPC) is emerging as the go-to platform for enterprises and academia looking to gain deep insights into areas as diverse as genomics, computational chemistry, financial risk modeling and seismic imaging. Initially embraced only by research scientists who needed to perform complex mathematical calculations, since the 2010s HPC has been gaining more and more the attention of a wider number of enterprises spanning an array of fields.

"Environments that thrive on the collection, analysis and distribution of data – and depend on reliable systems to support streamlined workflow with immense computational power – *need HPC*," is the leitmotiv that has been globally accepted by HPC data-storage-systems providers.

Whether adopted by small- and medium-size enterprises, by huge corporates, or by research institutes, HPC technology holds great potential for those organizations that are willing to make the investment. Most use cases focus on simulation, that is, modelling a complex real case scenario and trying to reason on the most suitable solution from a HPC point of view.

In this report we will follow a similar approach, providing a comprehensive analysis of many different implementations of Conway's Game of Life trying to identify the optimum approach from a HPC standpoint. In Section 2 we will give an introduction of Conway's Game of Life, and the multitude of interpretations that it has been attributed with ever since its introduction. In Section 3 we will discuss the design choices that we followed to carry out all the implementations, as such: serial, OpenMP, MPI and then CUDA. In Section 4 we will present all the experiments, clarifying the different setups, showing the relevant plots and discussing the gathered insights. Finally, in Section 5 we will summarize the conclusions with a few take-on messages from the Game of Life.

*All authors belong to the DIBRIS department of the University of Genoa.

Preprint. Under review.

2 Game of Life

Conway's Game of Life is defined as a cellular automaton[14] and it was invented by John Horton Conway in 1970. It is considered to be a zero-player game, which means that its evolution is completely determined by the initial state, and hence requires no further input. It is a very interesting problem from a theoretical point of view because it has the same potentiality of a Turing machine [8]. For more details on the relation between Game of Life and Touring machine consult Section 2.3.

In this project we present Conway's Game of Life as *i)* an example of cellular automaton, *ii)* a tool to solve scientific problems, *iii)* a benchmark for testing High-performance Computing paradigms.

An idea of how to build an universal Turing machine with Life is to consider an infinitely large grid where cells, and their absence, which can be coded with 0 and 1, represent digital information. After much effort, it has been shown that it is possible to arrange the automaton in such a way that the cells interact to perform computations, and this results in an emulation of the universal Turing machine. What this basically means is that every problem that can be written as an algorithm, it can be computed using the Game of Life, and this is a very powerful characteristic.

Another interesting characteristic of Game of Life is its capability to simulate what happens to organisms when they are placed in close proximity to each other. This can be seen as a simulation of the behaviour of organisms that share the same environment. In this scenario the game provides an example of emergence and self-organization. So we can state that Game of Life is a simple representation of birth, death, development, and evolution in a population of living organisms, for example bacteria.

We defined Game of Life as a cellular automata, so let's now try to deeply understand what a cellular automata is. A cellular automata is a class of mathematical models where allot of equal/similar units interact locally with their neighbours. Every unit follows very easy rules that are applied in parallel for all units at once in every discrete timestep. The important fact in this model is not what the cells actually are but just how they behave in interaction with their neighbors. Since is a quite generic structure many have tried to classify it in a more rigorous manner. A first classification was given by Wolfram, which divide them in four categories:

- automata in which patterns generally stabilize into homogeneity
- automata in which patterns evolve into mostly stable or oscillating structures
- automata in which patterns evolve in a seemingly chaotic fashion
- automata in which patterns become extremely complex and may last for a long time, with stable local structures. This class is considered to be the one that can simulate an universal Turing machine

We can also consider another classification in two special types of cellular automata: reversible and totalistic. The *totalistic cellular automata* has the characteristic that each cell is represented by a number, and the value of a cell at a given time, depends only on the sum of the values of the cells in its neighborhood at the previous time. If we consider in the sum the value of the cell itself, then we have an *outer totalistic cellular automata*. We mention this class of cellular automata because Game of Life is representative of such of class. Infact outer totalistic cellular automata with the same Moore neighborhood[18] structure as Life are sometimes called *life-like* cellular automata.

2.1 Core assumptions

A main characteristic of Game of Life is to be *undecidable*, which means that given an initial pattern and a later pattern, no algorithm exists that can tell whether the later pattern is ever going to appear. This is a corollary of the *halting problem*[16], which is the problem of determining whether a given program will finish running or continue to run forever from an initial point.

The *Game* develops on an infinite two-dimensional orthogonal grid of square cells which we named *grid*. A cell is represented by a unique square on the grid, so from now on we are going to use this two terms interchangeably. The cells can have one of the two states: ALIVE, or 1, and DEAD, or 0.

Each cell it has a certain number neighbors, and they are commonly classified as *von Neumann neighborhood*[21] and *Moore neighborhood*. The first class is named after the founding cellular automaton theorist, and consists of the four orthogonally adjacent cells. The latter includes the von Neumann neighborhood, as well as the four diagonally adjacent cells, for a total of eight neighbors.

In this project we use Moore's neighborhood to determine the state of the cell at the next timestep. For each cell and his neighborhood we have 2^9 (512) possible patterns, and each pattern determines if the considered cell will die or lives to the next timestep. In a two-dimensional system with a Moore neighborhood, the total number of automata possible would be 2^{2^9}, or 1.34×10^{154}. The above result justify why the Game of Life is such an interesting model from a High-performance Computing point of view.

Each cell start in the same state, except for a finite number that starts in a different state. The assignment of state values to the cells is called a *configuration*. The grid status evolve at regular time intervals, and the next status of the grid is computed using the current one.

2.2 Rules

One of the most interesting aspects of the Game of Life is the simplicity of the rules on which it is based, and how from such simple rules can emerge very complex behaviours and patterns.
Despite its simplicity, the system achieves an impressive diversity of behavior, fluctuating between apparent randomness and order. The Game of Life is a simple example of what is sometimes defined as *emergent complexity* of *self-organizing systems* [6], and it is the study of how elaborate patterns and behaviours can emerge from very simple rules.

The rules are defined as follows:

- Any ALIVE cell with fewer that two ALIVE neighbours dies, as if by under-population;
- Any ALIVE cell with two or three ALIVE neighbours lives on the next generation;
- Any ALIVE cell with more than three ALIVE neighbours dies, as if by overpopulation;
- Any DEAD cell with exactly three ALIVE neighbours sees its birth, as if by reproduction.

The above rules were chosen by Conway to meet the following criteria:

- There should be no growth spurt in the population;
- There should exist a few initial patterns with chaotic, ringing or unpredictable outcomes;
- There should be potential for *von Neumann's universal constructor* [17];
- The rules should be as simple as possible, whilst adhering to the above constraints.

2.3 Turing completeness

Building on what has been briefly stated early in Section 2, we try to better understand what's under the hood of the *Turing machine-Game of Life* parallelism. In order to do that, we first have to introduce a well-known cellular pattern, known as *glider*, Fig. 1. The glider was first discovered by R. K. Guy in 1970, and represents the smallest obtainable spaceship [13] inside Game of Life's pseudo-infinite world, able to travel diagonally at a speed of one cell every four generation, or $c/4$.

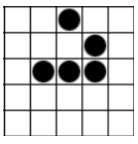


Figure 1: A simple 2D glider

Gliders play a key role within Game of Life's ecosystem, since they are easily reproducible, can collide with each other to form more complicated objects, and can be used to transmit information over long distances. For instance as few as just two gliders can synthesize complex patterns, such as blocks, beehives or blinkers, whereas as much as eight gliders can be positioned in a precise configuration such that by collision they form a *Gospel glider gun* [12]. The more complex the desired patterns the larger the required number of gliders (up to hundreds, thousands, or even millions in rare cases), i.e., by using the correct amount of gliders it is possible to construct logic gates such as AND, OR, and NOT which can inter-operate as a finite state machine.

All of the above remarks the aforementioned computational power of a universal Turing machine, that is, the Game of Life is theoretically as powerful as any computer with unlimited memory and no time constraints: *it is Turing complete* [20]! Even more surprisingly, as [22] shows, with a powerful enough machine it would be possible to reproduce Life inside Life infinitely many times like a fractal.

2.4 Structured grids

An important classification of applications with respect to the realm of parallel computing appeared in 2006 under the name of Berkeley's "dwarfs" [1]. A dwarf is an equivalence class into which applications are grouped based on their computation and communication patterns. Intuitively, parallel applications' run-time is split between a certain percentage of time spent performing calculations, and another percentage spent communicating those results. Dwarfs are able capture the general trends of these percentages, and classify parallel applications accordingly.

Any cellular automaton, Game of Life included, belongs to the *Structured Grid* dwarf, Fig. 2, wherein applications are characterized by the construction of points in a grid following a regular process, each of which are updated in close conjunction to the points around them. Structured grids are furnished with geometric (coordinates) and topological information (neighbour relations) which can be actively derived, without requiring them to be stored. As such, different classes of applications are compliant with such a definition, and can generally be considered to belong to either:

- *Cartesian/rectangular grids*, further split in rectangles (2D) or cuboids (3D);
- *Triangular meshes*, further split in triangles (2D) or tetrahedra (3D).

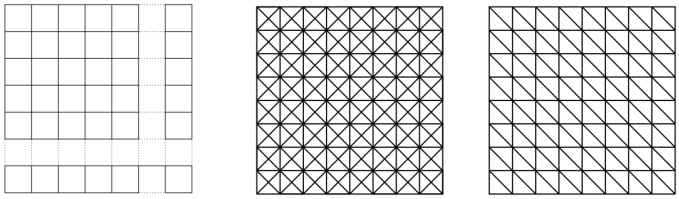


Figure 2: Regular structured grids

Whatever class an application may belong to, communication plays a key role and can be generally observed in multiple different flavors, i.e., direct-neighbour with ghost cells (described in greater detail in Section 3.6.2), diagonal communication with a 2-step communication schema, etc. In order to ensure a communication schema is sufficiently robust, though, neighbourhood weights have to be taken into account as well: depending on the application, different neighbours may have a different amount of influence based on how well they satisfy a certain condition, for instance distance-based. Many different strategies have been proposed over the years to assign meaningful weights to neighbours, but of all of them *Stencil computations* represent a well-known and effective solution: one of the most common stencils is the 2D Moore neighbourhood mentioned before in Section 2.1, which is indeed the communication schema of choice in the Game of Life.

It is worth noting that structured grids not only have a sound theoretical foundation behind them, but they also have strong practical implications. The heat model flowing across 2D grid boundaries, for instance, is constructed as a structured grid and solved via finite-difference methods (FDM) [15].

As we have already clarified in Section 2.1 and 2.2, in the specific case of the Game of Life grid points represent cells, whose update varies in relation to their neighbourhoods; hence, cell communication has to occur between each update of the GoL's grid, and each execution flow must communicate with the execution flows responsible for all neighbour cells.

But why are dwarfs relevant? Since all applications belonging to the very same dwarf have roughly the same computation and communication patterns, it is likely that the results of our scaling exercise (presented in Section 4) could be generalized to all the applications in the dwarf. In other words, if another structured grid application was to be scaled in the very same manner to what is presented here, the results on both Amdahl's and Gustafson's laws would likely be similar.

3 Design choices

In this section we will clarify many of the common design choices that we made approaching the problem of a serial to parallel implementation of the Game of Life, which will then expand in greater detail for each separate implementation, namely: serial, OpenMP, MPI, hybrid, and CUDA.

The most basic algorithm that we came up with to implement the Game of Life was to represent the cells with two-dimensional (2D) arrays in computer memory. Actually, two arrays of equal size proved to be even more valuable: one to hold the current generation, and one to store the next generation as it was being computed during the evolution. About the latter, the approach was again pretty straightforward: a simple loop taking turns considering each element of the current array, and for each of those counting the amount of ALIVE cells in the neighbourhood to establish whether the corresponding cell in the successor array should have been either ALIVE or DEAD.

Being the Game of Life such a widely studied topic, we got to know, or thought of, a bunch of possible improvements that we could have applied to this methodology, but did not have the practical time to test effectively. The very first improvement to the above algorithm, for instance, would be to not only keep track of the cells' state at the current generation, but also of their self state and their neighbours' at the previous generation with an additional data structure. Indeed if neither of those did change from the previous generation to the current, there would be no need to perform any computation because they would be unaffected in the next generation as well. By introducing such a tweak, we would be able to discriminate between inactive and active areas in the grid in order to gain in computational power by not updating the former.

We would also advocate two other possible directions to follow. Another improvement to the algorithm in fact would be to rearrange the rules from an egocentric approach of the cells' neighbourhood inner field to a scientific observer's viewpoint, defined as follows: if the sum of all nine fields in a given neighbourhood is 3, the inner field state for the next generation will be ALIVE; if the all-field sum is 4, the inner generation retains its current state; whereas every other sum sets the inner field to DEAD.

From a storage perspective instead, we could reduce the memory occupancy by using a single 2D array and two line buffers: one line buffer to calculate th successor state for a given row, and one to calculate the successor state for the next row. By doing so we would relax the need of a second duplicate 2D to store the whole successor state, which, as we will see with CUDA in Section 3.8, could be quite crucial in the maximum world size processable by our algorithm.

3.1 Custom data structures

In order to ease our understanding of the problems inherent within the Game of Life, we defined two custom data structures, *life_t* and *chunk_t*[2]. The former comprises all the necessary data required to handle a single-process Game of Life's instance, and it is heterogeneous enough to handle both CPU-based and GPU-based implementations by means of C guards pinpointing the subtle differences that those two different flavors would inevitably present; the latter instead is tailored for a multi-process setting wherein processes function as separate entities working on exclusive pieces of the Game of Life grid while in need of coordinating and communicating, as it is better explained in Section 3.6.

In particular, the *life_t* structure retains among its variables the overall number of columns and rows constituting the Game of Life grid, the overall number of generations to launch the simulation through, the initial probability of any given cell to be ALIVE to launch the simulation in a never stale pseudo-random setting (while still leaving room for the possibility to repeat the very same runs by fixing the random seed to a desired value), and the overall current and successor Game of Life grids. Also, in case OpenMP was enabled, it would retain the number of OpenMP threads each process would have to spawn; conversely, if CUDA was enabled, it would retain the block size, that is the amount of threads per CUDA block, as better explained in Section 3.8. It is important to note indeed that we never took into consideration the possibility of mixing OpenMP with CUDA between host and devices, thus this setup was already more than enough for our needs.

On the other hand, the *chunk_t* structure retains among its variables the overall number of columns and rows constituting the specific slice of the Game of Life grid assigned to the calling process,

[2]For a more thorough understanding of how we handled the various implementations into an all-in-one package, please refer to our GitHub repository at https://github.com/DiTo97/gol

its rank inside the MPI communicator, the overall number of running processes in the latter, the amount of leftover rows that had to be unevenly assigned to some of the processes, and of course the appropriate current and successor slices of the Game of Life's grid.

3.2 Default values

Despite us running extensive experiments (as shown in Section 4) on a plethora of different configurations of the Game of Life, we still opted to fix a few of the most characterizing default values that would look reasonable to construct the most generic configuration possible. In particular:

- The number of generation was set to 100;
- The GoL grid was set to a 50×50 matrix;
- The initial probability was set to 0.5, with a random seed of 1;
- The number of per-process OpenMP threads was set to 4, with a possible maximum value of 256 as a result of the Ocapie's cluster comprising 64-core-processors with 4 threads each;
- The block size was set to 128, with a possible maximum value of 1024, as per CUDA 7.5 specifications following the CUDA version installed on the GPU-capable cluster.

All of these values can still be manipulated at run-time via input arguments.

3.3 Grid swapping vs. Grid copying

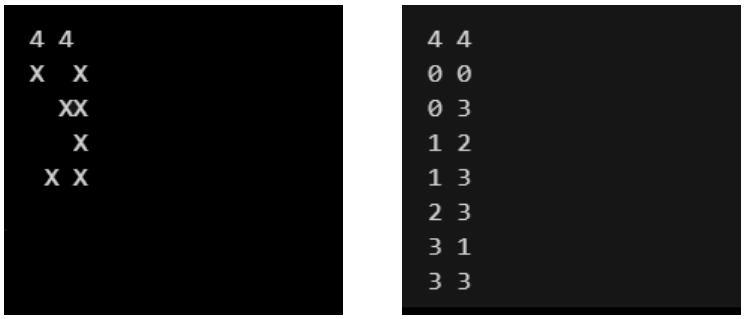
Expanding on the methodology presented earlier in Section 3, when dealing with two separate 2D structures to address the update each evolution brings to the Game of Life, at the end of each evolutionary stage it is important to re-assess the main 2D matrix with all the updated values out of the current evolution. Early in our exploration of the Game of Life, the simplest solution appeared to be a straight deep copy of all the cells of the successor matrix into the current one's before starting out with the subsequent evolution. This approach definitely had its merits and could also be parallelized, say with a *parallel for* OpenMP pragma; still, it was susceptible to the world size and the computation would inevitably take longer the larger the Game of Life grid was.

In order to circumnavigate this limitation, we opted to turn this step into a simple pointers swap between the two grids. This idea came only after we noted that, during the evolution, the algorithm was completely unaffected by the values present in the successor grid; hence, even if it retained the old grid values after the pointers swap, this would have been a non-issue. Of course, this approach has a few downsides of its own, with the most noticeable being the requirement of the whole 2D matrix to be stored into a contiguous section of memory. Since we didn't want to change a lot of the logic, which was based on the computation of a 2D matrix, rather than of a 1D array, but we still wanted to gain the benefits of this other approach, we opted for an in-between solution: during the initialization stage, the Game of Life grid (and its successor) was all allocated as a 1D matrix, but it was still made accessible as a 2D matrix, by making the matrix entries point to the array. In our opinion this got the best out of the two worlds, and was the best possible solution.

3.3.1 Full-matrix vs CSO display format

Another small feature that we included is the possibility to initialize the Game of Life grid from file. Since printing the resulting grid to file if it was too big was already a requirement, we opted to standardize the input and output format to the very same. In doing so we long discussed the original display format, which we called *Full-matrix*, or *FM*, that had a line for each row in the grid displayed with an *X* or a *' '* for each ALIVE or DEAD cell, respectively. Since the Game of Life grid can be generally thought of as a sparse matrix on various levels, we decided to experiment with a compressed format that would express line-by-line only the coordinates of cells marked as ALIVE, which took the name of *Compressed Sparse coOrdinate*, or *CSO*.

Why did we choose the CSO format initially? The CSO format facilitates parallel writing, at the expenses of parallel reading. Indeed, as the *printf()* function is thread-safe, we could guarantee that every single coordinates row was printed to file successfully even with multiple threads, kindly offered by OpenMP's pragmas. At that point, favouring the writing stage seemed more beneficial since, given a big enough matrix ($> 50 \times 50$), a print to file operation is always called both before and after all evolutionary generations have taken place in the code. By using this format, we would



(a) Full-matrix (b) CSO

Figure 3: Display formats on a 4×4 grid

sparsely save only the coordinates of cells marked as ALIVE, which we thought would save up both time (# of print operations) and space (file size in bytes).

Why did we discard it in favor of the FM format eventually? In practice, we found out that the run-time to print in CSO format would be skyrocket high with respect to FM's, because unless a lot of sparsity (just about 10% of ALIVE cells) was enforced in the grid, the # of print operations was exponentially higher, and the resulting files were around twice as large in byte-size. Originally, we thought that we would need to enforce no more than a 33% of sparsity in the matrix, which seemed reasonable, but our calculations proved to be wrong. And unfortunately, a 10% of sparsity was too much of a limiting factor to even consider CSO as a sensible display format.

Of course this discussion is relevant only in case the Game of Life is too big. Initially the evaluation of it being big or not was thresholded at a size of 100×100, but early experiments showed that displaying the grid to console is reasonable only for really low dimensions ($\leq 50 \times 50$). Otherwise, any bigger value will result in a visualization overflowing the console, thus rather meaningless.

3.4 Serial

As a starting point to the parallel implementation of the Game of Life, we started off with a serial approach where each evolution stage would comprise all the cells being processed iteratively. The main logic behind this implementation follows a skeleton that all the other below-presented implementations more or less adhere to.

First, the algorithm parses all the arguments, verifies whether default values should be retained or not, and constructs a self-contained *life_t* instance accordingly. Secondly, it initializes the Game of Life grids either randomly or from file, depending on what's been requested. This step is particularly crucial for obvious reasons, but it is also interesting to note that the whole initialization is performed sequentially, either by generating one random cell at a time, or by scanning the input file line by line; hence, we have left quite some room for parallelization even at this stage.

What follows afterwards is the evolution logic. At each new generation all the cells are updated by closeness, that is, depending on the information carried by their neighbourhood. Intuitively, we scan the whole Game of Life grid row by row (but column by column wouldn't really make any difference whatsoever) and for each cell we look for the 8 cells belonging to its Moore neighbourhood by applying some simple math to its 2D coordinates to avoid ending up out of bounds. Practically, we simulate an infinite world by ensuring that the rightmost (the top) cells are counted as the neighbours of the leftmost (the bottom) cells via the modulo operator, $\mod$.

Eventually, as soon as the whole grid has been updated, the successor grid gets swapped with the current one and a new generation can start right away. On a side note, printing operations have been deliberately omitted from this discussion for a few simple reasons: they have just been covered more or less in Section 3.3.1 and, more importantly, they are not subject to any relevant modification across all the different implementations. Indeed printing to either console or file has been voluntarily kept serial even in parallel settings to avoid the needs of any form of synchronization to show a consistent output that would be instead required if multiple sources tried to access the same stream.

3.5 OpenMP

Expanding on the few holes left deliberately open in the base skeleton, we observed that many sequential operations in the pipeline were actually independent from each other; hence, parallelizable. A small portion of this parallelism is omitted, as it would regard the parallelization of the initialization; despite that, the resulting speedup has been addressed in Section 4 nevertheless, by using the total run-time as a proxy to the evaluation of all these surrogate operations.

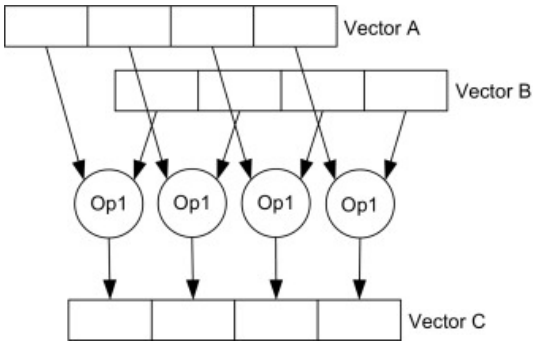


Figure 4: An example of SIMD

Whenever parallelism comes in small chunks, that is, a program is broken down to a large number of small low-demanding tasks that may repeat themselves over and over and over within a pipeline, we talk about data parallelism or *fine-grained parallelism*. This computing paradigm is known as *Single Instruction Multiple Data*, or *SIMD* [2], and relies on lots of data points being treated identically, that is, the very same instruction is performed a number of times on multiple data elements. Despite SIMD units being available in Intel microprocessors since the advent of the AVX extension in 2008, a computer will not catalog entire complex instructions as a standalone entity; hence programmers may need to restructure their code a little in order to allow for an efficient SIMD execution.

As each task processes less data, the number of processors required to perform the complete processing tends to be quite high. Consequently this increases the communication and synchronization overhead. That's mainly why fine-grained parallelism is best exploited in architectures which support fast communication, for instance shared memory architectures that present a low communication overhead.

OpenMP helped us solve all the above issues with a very easy-to-use API comprising convenient pragmas extensible to our base skeleton's fully iterative procedures. Indeed the main parallelization target ought to be the evolution stage since the current grid, left-untouched for the whole procedure, represented the shared memory all the cells would pull from. Then as all of them had to compute their neighbourhood status independently, that whole procedure was parallelized with OpenMP threads via the *parallel for* pragma: each separate thread would take care of the neighbourhood search, first, and the neighbourhood status evaluation, secondly, of a single cell. Moreover, before we made the jump from grid copying to grid swapping, already explained in Section 3.3, we were utilizing OpenMP to fasten the process of copying the updated successor grid into the current one. But, of course, as soon as we switched to grid swapping we had no real benefit from OpenMP; hence, we just took it out.

3.6 MPI

3.6.1 MPI logic

Differently from the serial and OpenMP implementation, the MPI computation is distributed on multiple nodes. This means that we don't have a shared memory architecture, but every node has his own memory which cannot be accessed by other nodes. Hence we need an inter-process communication paradigm (i.e. message passing) in which each node computes a chunk of the entire grid and sends the required information to the nodes that need it. At the same time obviously each process has to fetch his missing information from the other nodes. In particular the required information is the values of the neighbors that are on a node which is different from the one on which the computation is made.

Before starting the implementation we have done an analysis part in which we had to consider the following problematics:

- data partitioning
- communication
- agglomeration
- mapping

For the data partitioning we had to decide which domain decomposition to adopt. The main possibilities were *row* or *block* decomposition. Taking into account the communication factor and the fact that we are using a ghost rows logic, we decided to adopt a row decomposition with a fixed number of rows for each node. The ghost rows logic is explained in Section 3.6.2. This choice allow us to reduce the communication costs because we have to exchange among the nodes only the top and bottom neighbors. The left and right neighbors are already present in the node because we are considering full rows. If instead we consider the block decomposition, we have to exchange also the left and right neighbors. This can be seen visually in Fig. 5.

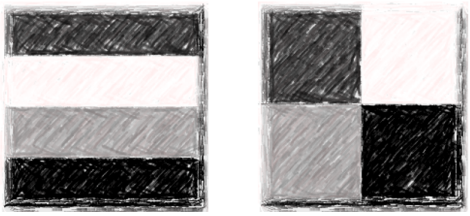


Figure 5: Row partitioning (left), block partitioning (right)

For what concern the agglomeration we decided to divide the number of rows equally among all available nodes, and to map starting from the top of the matrix, the first block to the first node, the second block to the second node, and so on. If the data does not equally split among the nodes we decided to associate the remaining data to the last node.

So considered the above information, each node has to compute his rows of the grid, send his firs row to the above node, and send the last row to the below one. By above node, we mean the node that has been assigned the rows that come before the first row in the current node, and by below node, we mean the node that has been assigned the rows that come after the last row of the current node. When we say first and last row we refer to the rows from the original data assigned to the node excluding the ghost rows. Then it has to receive the last row from the above node and insert it as the top ghost row, but also receive the first row from the below node and insert it as the bottom ghost row.

3.6.2 Ghost rows

Since the grid from a theoretical point of view is infinite, but at an implementation level we can only have a finite one, we had to simulate this theoretical aspect. We did it by using the concept of a grid with ghost rows. This approach can also be seen as a *torus* that wraps around on the top, bottom and sides. This also solves the problem of the neighbors for the elements which are on the edge of the grid. With this setting the cells on the top row have as neighbors the cells on the bottom row, and vice versa. The same applies for the left and right columns. The cells in the corners instead have one neighbour situated in the opposite corner (following the matrix diagonal). An image of the mentioned setting is presented in Fig. 6.

3.6.3 MPI implementation

We first defined the structure that each node will use to make the computation, as mentioned in Section 3.1. To better understand this structure, let's consider the following example:

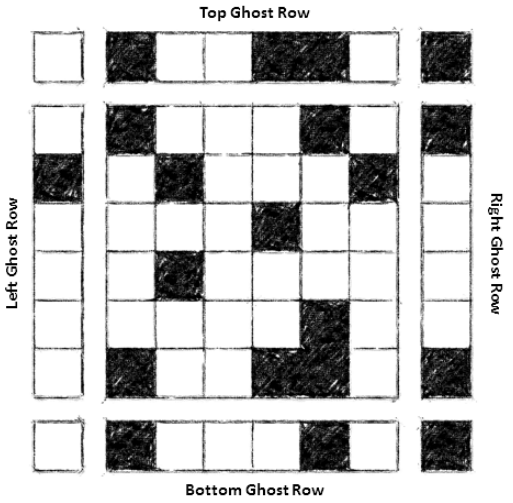


Figure 6: Grid with ghost rows

> *We have a grid which is composed by 500 rows, and we have 10 nodes to compute this grid. So each node has to compute 50 rows. Since we use ghost rows logic, we declared an array that will hold the 50 rows that the node has to process, plus two other rows which represent the top/down neighbors of that node.*

The two extra rows in the above example are needed because the node has to know what are the values for the row above his first row, and below his last row. One particular case that we had to consider was the case in which the division of the rows among the nodes was not even. As already mentioned, in this case we assigned the remaining rows to the last node. We did this by computing the index of the first and the last row that the node has to process, then we checked if the division was even or not. If the division is not even and we consider the last node, the index of the last row coincide with the index of the last row of the whole matrix. From a mathematical point of view we used the following formulations:

$$\#\_rows\_per\_process = \frac{\#\_total\_rows}{\#\_nodes} \tag{1}$$

$$displacement = \#\_total\_rows \,\%\, \#\_nodes \tag{2}$$

$$grid\_from\_index = node\_rank * \#\_rows\_per\_process \tag{3}$$

$$grid\_to\_index = (node\_rank + 1) * (\#rows\_per\_process - 1) + displacement \tag{4}$$

where *displacement* is always zero if the division in (1) is even, and different from zero only for the last node otherwise; *node_rank* is the rank of the process within MPI communicator; *grid_from_index* and *grid_to_index* are the indexes of the start and the end row attributed to the calling process, respectively.

After initializing all the required MPI variables, such as the communicator size and the communicator rank, etc., we compute the number of rows for each node, and also the initial and final index of the rows as mentioned above. Having this information allows us to create the arrays of grid chunks on each node. We then initialize each array with the appropriate values. In the initialization step we also let each process fill in the data corresponding to its initial ghost rows, from the previous and next node. By doing so we conveniently avoid the first round of message passing, as described in Section 3.6.5.

3.6.4 Chunk initialization

For what concerns the random initialization of the chunks we encountered some difficulties in generating the same sequence that we generated in the sequential case. The problem was that we

wanted each node to randomly initialize only his part of matrix, so that we avoid generating the whole matrix on one node and send the chunks to the others, which may end up clogging the former. The solution was to generate on each node the whole sequence of random numbers, and by whole we intend the sequence that we used to fill the matrix in the sequential case, but we saved only the values belonging to that particular node plus the two ghost rows. This is far more optimal then sending the chunks of the matrix from one node to the others because the generation of random numbers is a very lightweight process. Successively we improve the initialization by stopping the random generation after we had all the values that are needed for a particular node. For the first node we had to generate the whole sequence because we needed the values on the last row of the whole matrix since they represent the values for the top ghost row.

For what concern the initialization from file we used the same approach. All the nodes read the file, and save locally only the rows that are associated with the considered node.

3.6.5 Grid evolution

We now loop through the time steps, and each node computes his chunk of data. Since each node has a current and a next state of the chunk, we have to update at each iteration, the current chunk with the data of the next chunk. Considering the fact that each chunk is represented as an unidirectional array, as already mentioned in Section 3, what we did was to swap the pointers between current and next chunk in order to simulate the copy process. This approach is far more optimal from a computational point of view then copying the elements from a chunk to the other.

At this point given a certain node *n* we compute the node before and the node after. These nodes are needed to send and retrieve the ghost rows. From a mathematical point of view we use the next formulations:

$$prev\_node \ = \ (node\_rank - 1 + \#nodes) \ \% \ \#nodes \tag{5}$$

$$next\_node \ = \ (node\_rank + 1) \ \% \ \#nodes \tag{6}$$

which compute the ranks of the neighbor nodes inside the communicator.

The following is a schematic definition of the steps that we considered in the MPI implementation.

1. Node 0 parses the arguments and fills the struct life_t with the required values
2. All nodes store the number of processes and the communicator size
3. All nodes compute the indexes for for their chunk of data
4. All nodes allocate memory for their chunk of data in a chunk_t structure
5. All nodes initialize their chunk with random values or from file if present. The nodes also save the ghost rows from the other nodes.
6. For all the generations:
    - 6.1 All processes evolve their chunk
        - 6.1.1 If the grid is small:
            - 6.1.1.1 All nodes (except node 0) send their updated chunk to node 0
            - 6.1.1.2 Node 0 sequentially prints the updated grid to console
        - 6.1.2 If the grid is large, all processes exchange the ghost rows
7. All nodes (except node 0) send their final chunk back to node 0
8. Node 0 sequentially prints the final grid to console if small or save it on file if large

We exchange the information between nodes by using two MPI_Sendrecv functions, one for the top ghost row and one for the bottom one. The MPI_Sendrecv can be seen as two different operations:

- *MPI_Send* which sends a message to a give node

- *MPI_Recv* which receive a message from a given node

Usually the source and the destination in a MPI_Sendrcv are the same node, and the context in which it is most used is when we need to execute a shift operation across a chain of processes, which is our case. The advantage of using a MPI_Sendrecv function instead of calling an MPI_Send and an MPI_Recv is that it handles automatically the order of the send and receive operations in order to avoid cyclic dependencies that may lead to deadlocks. It is also more efficient form a performance point of view.

3.7 Hybrid

We build the hybrid implementation on top of the MPI one by enabling the OpenMP functionalities where possible. In particular from an architectural point of view, we split the initial grid over the available nodes and on each node instead of processing the cells using the MPI processes only, we use a combination of MPI processes and OpenMP threads. More precisely each MPI process has a certain number of threads allocated, which work in parallel to split the chunk of data associated with that particular process. This allows us to speedup the computation by using the shared memory and multi-thread processing capabilities of each node jointly.

3.8 CUDA

3.8.1 Memory coalescing

The GoL program parallelized with the NVIDIA CUDA paradigm makes use of **memory coalescing**, which is a technique that allows optimal usage of the GPU global memory bandwidth. The difference between coalesced and non-coalesced memory accesses is reported in Figure 7. In Figures 7.a and

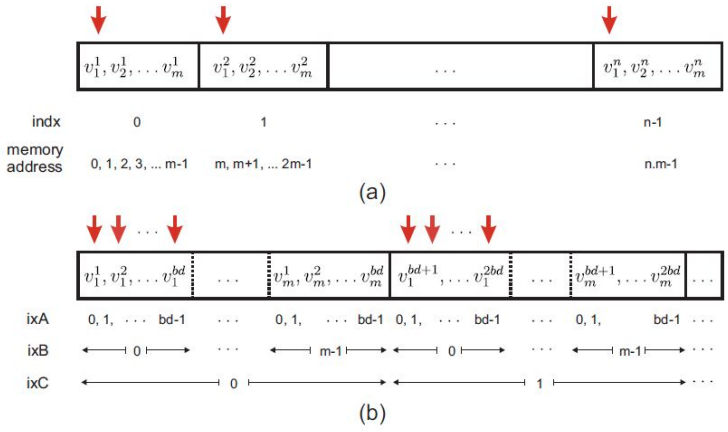


Figure 7: Storing n m-element vectors (a) in linear order (b) in coalesced pattern.

7.b, n vectors of length m are stored in a linear fashion. Element i of vector j is denoted by v_i^j, and each red arrow represents a different thread. The example in figure 7.a represents the case in which threads access only the first components of each vector, which are identified by memory indexes $0, m, 2m, ...$ In this case, we end up with a non-coalesced memory access, that is, a memory access which is not consecutive across different threads. To make the memory accesses coalesced, we have to consider that CUDA threads are organized in grids formed by blocks each comprising bd entries, and that every GPU thread is associated with a unique identifier, which also identifies one of those entries. Having in mind this organizational pattern, the coalescing technique is implemented by eliminating the gaps between the red arrows: the first bd elements of the first m-element row vector are stored in consecutive order, and are followed by the second bd elements of the same vector, and this schema is applied for all the m elements of each of the n vectors. The access pattern described here leads to improved performance because of the run-time division of CUDA blocks into **warps**, which we recall to be the smallest CUDA unit of execution. In particular, as explained in [3], [7], [4],[9], and [10], the GPU device groups global memory loads and stores issued by threads of a

warp into as few transactions as possible to minimize the amount of bandwidth usage, provided that your memory is contiguous and contiguous threads in warps access contiguous memory cells. In our case, the GPU used was an NVIDIA GeForce GTX 1650, whose compute capability is 7.5, hence transactions are grouped within full warps of 32 threads, differently from devices of compute capability lower than 2.0, which only use half-warps of 16 threads each. Therefore, the system performs a single transaction with the global memory for each warp, and since all the warps are executed simultaneously by the corresponding **streaming multiprocessor** (SM), we greatly reduce the total time which we need to retrieve data from the global GPU memory at each generation.

3.8.2 Implementation details

After having presented the main technique used to implement memory accesses, we now dig into the details of our CUDA-parallelized version of GoL. As one may already have guessed from the previous section, the CUDA-parallelized program uses a single array of contiguously allocated memory to encode the GoL board. This is done both from the CPU side and the GPU side, in order to ensure that the device hardware uses the memory coalescing access pattern. In particular, we allocate one array for the current status of the GoL board and another one for the resulting status of the GoL board after each generation. The allocation and initialization procedures of the device memory are performed by using **cudaMalloc** and **cudaMemcpy**. In particular, **cudaMalloc** allocates a certain number of bytes of **linear memory** on the device, and we pass this number as a single **size_t** integer value to enforce a contiguous allocation.
The actual implementation of each evolution step is performed by means of the CUDA kernel **evolve**. In particular, we enforced each kernel call to be associated with the computations concerning a single GoL board cell. This has been done by computing the number of blocks as a function of the number of threads per block, as follows:

$$N_B = \begin{cases} \lfloor \dfrac{(N_R \cdot N_C + N_{TB} - 1)}{N_{TB}} \rfloor & \text{if } (N_C \cdot N_R) \bmod N_{TB} \neq 0 \\ \lfloor \dfrac{(N_R \cdot N_C)}{N_{TB}} \rfloor & \text{otherwise} \end{cases} \tag{7}$$

where N_{TB} is the number of threads for each CUDA block, as specified by user input, whereas N_R and N_C are the number of rows and columns of the GoL board when viewed as a 2D array, respectively. Since threads are lightweight in CUDA, we therefore don't care about using more threads than required. Threads which are not mapped to cells are handled inside **evolve**, whose first step is to return immediately if a thread exceeds the boundaries of the 1D board.
The procedure executed in **evolve** makes use of an indexing strategy which allows us to compute the number of alive neighbors of a cell while using a 1D array as GoL board. The sum obtained this way is then used to determine whether the cell will die in the current generation or live in the next generation. The 1D array used stores rows of the 2D GoL board one after the other, so that the size of the contiguously allocated memory is $N_C \cdot N_R$ for both the current and successor 1D boards.
The indexing strategy starts with the computation of the index of the board cell associated with a CUDA thread:

$$C_{idx} = B_{idx} \cdot B_{dim} + T_{idx} \tag{8}$$

(8) is a standard formula which must be used when CUDA blocks are 1D: B_{idx} is the index of the block comprising thread T, itself identified by T_{idx} inside B, whereas B_{dim} is the total number of cells in the 1D block B. From (8), the algorithm derives

$$x = C_{idx} \bmod N_C \tag{9}$$

$$y = C_{idx} - x \tag{10}$$

Equations (9) and (10) represent, respectively, the index of the column containing cell C when we view the GoL board as a 2D array (which can also be called **zero-indexed column**) and the start of the row containing C in the global 1D array encoding the GoL board. As the next step of **evolve**, (9) is further used to compute the zero-indexed columns corresponding to the left and right neighbor of C, which are respectively defined as follows:

$$x_l = (x + N_C - 1) \bmod N_C \tag{11}$$

$$x_r = (x + 1) \bmod N_C \tag{12}$$

Equation (10) is then used to compute the start of the rows containing the upper and lower neighbors of C in the global 1D array encoding the GoL board. These two indexes are respectively obtained as:

$$y_u = (y + N_C \cdot N_R - N_C) \bmod (N_C \cdot N_R) \tag{13}$$
$$y_d = (y + N_C) \bmod (N_C \cdot N_R) \tag{14}$$

We then count the number of alive neighbors of C. The formulas used to obtain the indexes of the neighbors of C make use of all the equations defined so far to describe the CUDA kernel implementation, and are the following:

$$n_{ul} = x_l + y_u \quad \text{top-left neighbor} \tag{15}$$
$$n_u = x + y_u \quad \text{upper neighbor} \tag{16}$$
$$n_{ur} = x_r + y_u \quad \text{top-right neighbor} \tag{17}$$
$$n_l = x_l + y \quad \text{left neighbor} \tag{18}$$
$$n_r = x_r + y \quad \text{right neighbor} \tag{19}$$
$$n_{dl} = x_l + y_d \quad \text{bottom-left neighbor} \tag{20}$$
$$n_d = x + y_d \quad \text{lower neighbor} \tag{21}$$
$$n_{dr} = x_r + y_d \quad \text{bottom-right neighbor} \tag{22}$$

As a last step of **evolve**, we simply sum all the contents of the cells corresponding to global 1D array indexes (15), (16), (17), (18), (19), (20), (21), (22), since the GoL board contains boolean values to indicate alive cells and dead cells, and we check all the rules described in Section 2.2 with a single and compact if-else statement, implemented with the ternary operator $(?, :)$. The boolean formula which implements the GoL test is the following:

$$A_{C_{t+1}} = \begin{cases} 1 & \textit{if } (N_{alive}(C_t) = 3) \vee (N_{alive}(C_t) = 2 \wedge A(C_t) = 1) \\ 0 & \textit{otherwise} \end{cases} \tag{23}$$

where C_t represents a cell of the GoL board at generation t, whereas functions N_{alive} and A respectively count the number of alive neighbors of C_t and determine whether C_t is alive. The choice of the ternary operator is motivated by [11], in which it is discussed whether it should be used in place of the usual if-else statement to improve the performance of CUDA kernels. In general, using explicit if-else clauses results in the conversion of the CUDA C code into branched instructions based on a test of an expression, which may lead to **warp divergence** depending on the compute capability of the device hardware. Warp divergence causes threads of a CUDA warp to first execute all the tasks generated by one code path and then the ones associated with the other code path, resulting in great performance loss. In practice, the ternary operator ensures that warp divergence is correctly handled by CUDA in a way such that all the computations of both paths are executed in parallel, independently of the compute capability of the device. Moreover, the usage of (23) avoids the introduction of too much branching in the kernel code, as multiple conditions are tested with a single if-else statement, and hence further reduces the possibility of warp divergence.
In the end, the host program synchronizes with the device by calling the CUDA function **cudaDeviceSynchronize** after the completion of **evolve**. After each **cudaDeviceSynchronize** call, pointers referring to the old configuration of the GoL board and to the results of the current generation are swapped, for the purpose of initializing the execution of the next generation. Swapping has been preferred again over copying results for optimization purposes, similarly to the choice reported in Section 3.3: in this case, the requirement of having contiguously allocated memory has been fulfilled again by the usage of a 1D array to implement the GoL board.

3.8.3 About not taking advantage of per-block shared memory

The CUDA-parallelized GoL program does not make use of per-block shared memory in the kernel implementation, but directly optimizes the accesses to global memory by means of the memory coalescing pattern. We decided not to exploit per-block shared memory because, by construction of the kernel, each CUDA thread accesses a relatively small chunk of data which corresponds exactly to 9 memory locations comprising the cell to be updated and its neighbors, each 1 byte long, and since data is first searched in the block-level L1 cache, which is itself implemented into the shared memory as a non-controllable hardware with the same circuitry [9], and then in the L2 cache shared by multiple SMs, then the data is not necessarily evicted by subsequent global memory loads, and can be intelligently replaced by the hardware's internal policies as the program performs computations.

Therefore, in principle, we don't necessarily have to force the data to be located in the shared memory and hence we can save the effort of implementing custom logic to manage cached data. This can be more clearly understood by considering the technical specifications of the NVIDIA GeForce GTX 1650 GPU: the number of SM is 14, the L1 cache size can be occupied, at maximum, up to 48 kB for each SM, and this leads to an overall amount of L1 cache memory of $14 \cdot 48000 = 672000$, which if summed with the L2 cache size outputs $1048576 + 672000 = 1720576$ bytes. This means that, for example, when considering GoL boards of size $10000 \cdot 10000$, which is the maximum size we benchmarked for the serial GoL code, the number of accesses performed by CUDA threads to the global memory at each generation is $9 \cdot 10000 \cdot 10000 = 900000000$ in the worst case, and the number of accesses to the global memory for each transaction performed with the two caches combined, after they have been filled, is equal to $\frac{900000000}{1720576} \simeq 523$. This fraction is greatly reduced if we consider that we implemented the memory coalescing access pattern, which performs transactions to the global memory in groups of 32, and therefore the number of transactions performed with the global memory for each cache transaction becomes $\frac{523}{S_{warp}} = \frac{523}{32} \simeq 16$ in the worst case, a number not that big, considering the high dimensions of this example and that both L1 and L2 caches in GPUs usually implement clever replacement policies and both present much lower latency than the global memory.

4 Experiments

4.1 Setup

For the purpose of experimenting the Game of Life program, we performed $n \times m \times r$ for each binary corresponding to one of the five implementations, with n being the length of a list $B_{dims} = [100, 500, 1000, 5000, 10000, 20000, 40000]$ of board dimensions, m being the number of configurations of the parameter of choice for each implementation (e.g., threads for OpenMP) and r being the number of repetitions that we conducted each experiment for - in this case set to 3.

Each board dimension from B_{dims} was treated as the side of a square Game of Life's grid for simplicity, since the shape of the grid did not prove to be relevant for the purposes of our discussion. Instead the rationale behind running a number of repetitions greater than 1 for each experiment was to regularize the results and achieve greater statistical stability.

Moreover, each individual run of any experiment on the Game of Life was let evolve for a grand total of 100 generations. Indeed since our approaches do not make use of any advanced technique to coalesce the attention of the evolution only on *active* areas wherein some actual evolution would have to happen somehow, the effort to move from generation t_i to generation t_{i+1} is more or less the same, regardless of the amount of requested generations; hence, we reduced the number of generations from 1000 to 100, since we thought running the extra generations wouldn't have added any extra information to our analysis, and 100 was already a big enough number not to falsify the results if a given generation was mistakenly slower than the others for whatever reason.

The other parameters were left as default, as presented in Section 3.2. In particular, in order to make the experiments reproducible across multiple runs, we left the random seed as 1, so that all the implementations could be tested on the very same Game of Life's grid. Again, since the sparsity of the matrix does not cause any difference in the approach of our algorithm, which would still visit all the cells, testing different topologies of the grid was not relevant for the purposes of our discussion.

In each run, we kept track of two metrics:

- **Cumulative generation time**, defined as the sum of the execution times of the evolution stages as the most demanding stages of the algorithm bulk-wise.
- **Total program time**, which includes the former, as well as, arguments parsing, initialization of all the Game of Life's structures, update of the GoL's grids across evolutions, results printing to either console or file, and cleanup operations.

Differentiating the two metrics served us perfectly. Indeed by analyzing the effects of parallelization on separate levels, we were able to observe its benefits in full-display in the evolution, as well as its possible drawbacks in the need of synchronization to make everything else work as intended.
In this context then, we would have to expect far higher speedups, with respect to the serial base case, registered with the former metric rather than with the latter.

In the end, for each pair (a, b), with a being a particular board dimension from B_{dims}, and b being a given configuration of the relevant parameter, we averaged all the corresponding r values for both of the metrics described above. All the tables and plots reported in Section 4.2 are the result of considerations derived by comparison of said metrics.

4.1.1 Serial

In experimenting the serial implementation we decided to start from the most bare-bone version, and gradually introduced vectorization, via compiler optimization, offered by Intel's ICC compiler.

In this regard we opted to always adopt the *-xHost* and the *-ipo* flags, because the former never introduced any significant difference between having it on or not, whereas the latter proved to trade a slower compilation for a better run-time performance, which was a favourable trade to take.

About the main optimization flags instead, we tried them all, from *-O0* to *-O3*, but discarded *-O1* and *-O3* early for different reasons: the former, tailored for optimization for code size, never managed to shrink the binary size down substantially, while always being 4 to 5 times slower on average than the best optimization level, *-O2*, under every condition of the Game of Life grid, either small, medium or large, and either dense or sparse; the latter instead proved to always be slightly inferior to *-O2*, around 1.1 to 1.3 times slower on average, despite being listed as tailored for the maximum optimization for speed. In fact, this seems to be quite a common pattern, and mainly related to the quality of most non production-ready code.

This left us with both *-O0* and *-O2* to be tested extensively, the former producing the most bare-bone binary possible without any sort of optimization, and the latter proving to be the best all around optimization level on a single machine. In this context, fixing the total number of generations to 100 proved to be a life-saver, as the slowness of the serial implementation proved to be a serious bottleneck to the fulfillment of out experiments on the Ocapie cluster.

4.1.2 OpenMP

As we have detailed in Section 3.5, there are no real differences between the serial and OpenMP implementations, besides the use of threads itself which exploits simultaneous multi-threading (SMT) [19]. Then, to add value to our experiments with OpenMP, we decided to identify which configurations of threads would be the most meaningful with respect to the Ocapie cluster's specs.

In fact, we ended up with a good lot of OpenMP threads configurations following the powers of 2 to combine with all the different board sizes, as such: [2, 4, 8, 16, 32, 64, 128, 256]

Many of those had a very specific rationale behind them, exploring SMT. On a single node, for instance, *256* would represent the node's total processing capability (as we recall from Section 3.2 each Ocapie cluster's node is equipped with one CPU, each with 64 cores and 4 threads per core); *64* would represent the node's total physical processing units, that is, again the 64 cores above; *32* would represent instead half of total physical processing units, so to make each thread exploit the L2 cache, shared among multiple threads, more efficiently by reducing the number of global memory accesses; *2* would represent the smallest parallelization units for the Game of Life.

It also interesting to note that, out of curiosity, we tried to run OpenMP parallelization with just 1 thread, which we expected to have similar performances to the serial implementation. In fact, actually, despite not being present in the charts in Section 2, it always showed to be quite faster than its serial counterpart, possibly due to OpenMP's internal optimizations.

4.1.3 MPI

The main scenarios that we considered in the MPI experiments are *single node execution*, *2-node execution*, *4-node execution* and *8-node execution*. We considered the single node execution because we wanted to make a comparison with the serial case, since the MPI execution on one node is in fact a serial execution using the MPI infrastructure. Two nodes is the minimum level of nodes parallelism and it can be considered the base line for the MPI execution. Then we run the program on four nodes to see how the performance increase if we double the number of nodes. The idea for the last experiment was to run it with the full capabilities of the cluster, but since we noticed that it would require quite some time to complete a set of experiments, and since we were not the only ones that used the cluster, we decide to settle with eight nodes.

Adding onto the parameters that we have already discussed in Section 4.1, we also utilized a board dimension of 10000. This value was not used to create a square matrix, though, but constructed a grid with the specific topology of 1000 rows and 10000 columns. The rationale behind this experiment was to test the MPI implementation, which makes use of row-splitting to form blocks, in as setting where processes would have to process quite a few more cells in their chunk. In fact, this experiment did not show any significant difference from its square counterpart.

The parameter that we varied along with the number of nodes is the *number of processes* that had to be spawned. This was required since the number of processes available increases linearly with the number of nodes. In particular we had:

- single node: [2, 4, 8, 16, 32, 64, 128, 256]
- 2-node: [2, 4, 8, 16, 32, 64, 128, 256, 512]
- 4- and 8-node: [2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]

The above values are considered in nested for loops, so the experiments consist in all the possible combinations of the number of repetitions, board dimension and number of processes.

To run all the above combinations of parameters we used the arguments in the following command:

```
mpiexec -hostfile nodes_list_file -perhost processes_per_node \
-np tot_num_processes,
```

where the *nodes_list_file* is a file which contains the names of the nodes on which the program can run, the *process_per_node* indicates the number of processes to be executed on each node and *tot_num_processes* indicates the total number of processes to be executed globally. If we also add the *-nolocal* flag then we require that on the node on which we launched the program no computation should be made.

4.1.4 Hybrid

In case of the Hybrid architecture, the experiments were run by using a number of nodes equal to 8 and 1. The single node execution case was considered because we wanted to find the minimum performance achievable when fully exploiting the computational capabilities of the combination OpenMP-MPI. The 8-nodes execution corresponds, as per the MPI experiments, to the case in which we try to exploit the full computational capabilities of the cluster, as we distribute the workload on multiple nodes and at the same time divide computations across multiple logical and physical cores. For what concerns all the other values used for the experiments, in this case we used the same setting as in the previous section concerning MPI, with the difference that we also consider the number of OpenMP threads that we run for each process. So in this case we have an additional loop which considers the number of threads with the following values: [2, 4, 8, 16, 32, 64, 128, 256]. The OpenMP threads, as well as the MPI processes, work on the virtual cores of the underlying machine. As we have already clarified multiple times, Ocapie's cluster nodes were equipped with 256 virtual cores, each, thus in order to properly test the hybrid implementation we had to enforce that the number of OpenMP threads and MPI processes satisfied the following relation:

$$\#openmp\_threads * \#mpi\_processes = 256 \tag{24}$$

The above formulation is considered for each single node.

Changing the number of threads for each process, allowed us to consider different edge cases for the single node. For example if we use up to four threads per process this means that they share the L1 cache level memory, from five to eight threads per process means that they share the L2 cache, more than eight means that they share the global memory, and so on.

4.1.5 CUDA

The CUDA-parallelized GoL program has been tested on a single machine of an HPC cluster formed by a total of 10 nodes. The machine was equipped with an NVIDIA GeForce GTX 1650,

whose compute capability is 7.5. The experiments performed with the CUDA binary were the same as the ones reported in Section 4.1, with the parameter m chosen to be the length of a list $B_{sizes} = [32, 64, 128, 256, 512, 1024]$ of N_{TB} values (see Section 3.8 for the definition of N_{TB}). In this case, therefore, configuration b corresponds to a value of N_{TB}. Each d element of B_{dims} was used to initialize a contiguous 1D GoL board having $d \times d$ cells, as already described in Section 3.8, for the purpose of implementing the memory coalescing access pattern. The motivations for choosing the values in B_{sizes} are the following:

- 32 has been chosen as the smallest possible value for N_{TB} to perform the CUDA experiments, because it corresponds to the dimension of a warp in hardware devices having compute capability 7.5, and since a warp is the smallest CUDA unit of execution, a value of N_{TB} smaller than 32 would lead to a highly inefficient parallelization.
- 1024 has been chosen as the maximum possible value for N_{TB} to perform the CUDA experiments, because it corresponds to the maximum number of threads per block supported by the hardware of the NVIDIA GeForce GTX 1650 GPU.
- Values between 32 and 1024 in list B_{sizes} have been chosen because they are all multiples of the warp size, which is 32. In fact, as pointed out in [3], setting a block size multiple of the warp size facilitates memory accesses by warps that are properly aligned: in particular, in CUDA GPUs having compute capability 6.0 or higher, memory allocated through **cudaMalloc** is guaranteed to be aligned to at least 256 bytes, and misaligned accesses of a single warp lead to the execution of a single load transaction of 5 32-byte segments, saving execution time for the transactions performed by other adjacent warps, as they request slices of memory included in the aforementioned 256-long segment and hereafter allow the hardware to better exploit the L1 and L2 caches and minimize the bandwidth usage of the global memory.

For what concerns the metrics already mentioned in Section 4.1, the cumulative generation time was defined as the sum of the execution times of the **evolve** kernel calls, whereas the total program execution time included command line arguments parsing, random GoL board initialization on the host side, data transfer from host to device to setup the initial status of current and next generation grids and data structures cleanup on both the host and device side. As per all the other binaries, for each pair (d, N_{TB}), we in the end averaged all the corresponding values for both the metrics described before.

4.2 Results

We now present the results of the experiments described in Section 4.1. This section is divided in two parts: one containing the results for the cumulative generation time, and the other the ones for the total program time. Each of these subsections reports, in the following order:

1. The execution times in milliseconds of the GoL serial baseline used to perform the speedup computations, computed with the metric corresponding to the subsection name.
2. One after the other, the speedup tables computed for the vectorized, OpenMP-parallelized, MPI-parallelized, Hybrid OpenMP-MPI, and CUDA-parallelized programs. Each speedup value is computed with the already described formula $\frac{T_s}{T_p}$, and with respect to the pair (d_1, d_2), where d_1 is one among the values for the size of the GoL board reported in the table described at 1. (in which it goes by the name of *World size*), and $d_2 = d_1$ is the corresponding board size value in the table reporting the execution times of the parallel program currently being considered.

The parallel program data which have been used to compute the speedup values are reported in appendix A, which contains the execution times in milliseconds obtained in all the experiments described in Section 4.1. Appendix A also reports execution times obtained as a result of further experiments, which have been performed by running all the parallel programs with board sizes larger than $d = 10000 \cdot 10000$: we included them in the appendix for the sake of completeness of the analysis to be performed, as well as for time constraints, which impeded us to run the serial program on board sizes $40000 \cdot 40000$ and $20000 \cdot 20000$ and hence to compute the speedup values for these two dimensions.

4.2.1 Cumulative generation time

Serial baseline

Table 1: Cumulative generation time in milliseconds - Serial

	O0
10000	550.76
250000	13328.2
1000000	53465.8
25000000	1.471e+06
100000000	5.351e+06

Speedup: Serial baseline

Table 2: Cumulative generation time speedup - Serial with optimizations

	O2	O2+xHost
10000	5.27	5.27
250000	**5.16**	5.15
1000000	5.16	5.16
25000000	5.71	5.71
100000000	**5.19**	5.18

Table 3: Cumulative generation time speedup - OpenMP

World size	Total # of threads							
	2	4	8	16	32	64	128	256
10000	10.07	17.10	32.40	67.91	111.94	182.98	**188.62**	12.90
250000	9.96	24.07	47.67	93.41	157.97	272.12	**340.44**	307.53
1000000	10.48	23.52	40.02	79.71	157.44	269.54	**345.88**	305.59
25000000	11.81	24.12	49.22	105.75	211.59	296.49	**393.17**	374.79
100000000	10.80	22.10	41.30	95.20	192.89	271.90	357.43	**388.71**

Table 4: Cumulative generation time speedup - MPI - 1 node

World size	Total # of processes across nodes							
	2	4	8	16	32	64	128	256
10000	10.11	20.13	32.63	56.14	**67.66**	15.90	3.87	1.43
250000	10.58	21.91	45.57	**86.31**	85.90	51.24	16.22	4.27
1000000	10.71	22.21	48.70	87.06	**152.69**	111.15	34.87	10.21
25000000	11.72	23.78	52.39	104.92	202.73	262.82	**363.55**	83.87
100000000	10.64	21.49	47.89	97.02	185.91	239.01	330.56	**355.01**

Table 5: Cumulative generation time speedup - MPI - 2 nodes

World size	Total # of processes across nodes								
	2	4	8	16	32	64	128	256	512
10000	10.24	18.38	31.89	49.35	**63.38**	15.39	5.99	3.54	1.83
250000	10.50	20.91	42.36	**85.03**	84.84	50.81	25.33	7.69	2.37
1000000	10.54	21.04	42.13	84.96	**155.94**	110.31	54.86	16.24	4.85
25000000	11.67	23.29	46.64	98.16	205.37	390.33	**710.62**	136.04	31.74
100000000	10.62	21.17	42.53	93.31	186.98	355.38	648.72	**675.4**3	80.39

Table 6: Cumulative generation time speedup - MPI - 4 nodes

World size	Total # of processes across nodes									
	2	4	8	16	32	64	128	256	512	1024
10000	11.80	19.82	25.76	42.24	**59.35**	15.47	6.00	5.94	3.26	0.32
250000	12.18	20.99	28.69	58.98	**84.99**	50.87	25.39	12.34	3.78	2.04
1000000	12.21	21.18	42.20	59.85	**119.26**	110.75	54.22	25.26	7.75	2.31
25000000	13.46	23.43	46.72	67.60	132.13	264.28	**526.36**	216.39	51.86	14.07
100000000	12.24	21.23	42.56	61.82	123.54	240.30	480.05	**955.13**	131.77	29.34

Table 7: Cumulative generation time speedup - MPI - 8 nodes

World size	Total # of processes across nodes									
	2	4	8	16	32	64	128	256	512	1024
10000	11.90	19.69	30.87	41.50	**54.75**	13.52	5.95	5.58	5.75	3.17
250000	10.27	21.59	41.39	71.44	**84.55**	50.79	25.42	12.37	6.07	3.77
1000000	11.03	21.19	42.12	83.28	**140.63**	110.19	54.97	25.97	12.44	3.80
25000000	13.20	23.25	41.23	85.18	153.68	284.38	**527.76**	216.70	83.46	23.18
100000000	11.72	21.22	41.59	82.88	169.89	305.95	544.90	**957.23**	210.11	48.03

Table 8: Cumulative generation time speedup - Hybrid - 1 node

World size	Total # of processes × # of threads per process						
	2×128	4×64	8×32	16×16	32×8	64×4	128×2
10000	51.95	54.15	51.73	41.36	**67.22**	18.24	4.14
250000	**400.57**	399.20	288.51	318.15	197.11	67.89	17.63
1000000	**482.75**	469.05	165.28	402.76	400.90	146.18	37.84
25000000	537.33	**566.16**	553.30	556.40	538.43	512.48	472.93
100000000	503.43	**511.07**	507.66	508.90	498.80	476.81	396.46

Table 9: Cumulative generation time speedup - Hybrid - 8 nodes

World size	Total # of processes × # of threads per process							
	8×256	16×128	32×64	64×32	128×16	256×8	512×4	1024×2
10000	2.14	**13.01**	12.19	7.17	1.88	7.14	2.61	1.54
250000	128.35	**138.03**	38.95	10.94	20.32	12.37	8.20	4.16
1000000	418.13	**875.48**	150.37	195.97	48.36	50.42	16.79	4.16
25000000	1430.35	**1846.22**	1571.12	1601.04	1742.85	534.76	113.47	28.67
100000000	1183.96	1720.51	**2003.81**	1867.12	1885.27	1216.06	279.63	61.09

Table 10: Cumulative generation time speedup - CUDA

World size	Block size					
	32	64	128	256	512	1024
10000	834.48	1019.93	**1079.92**	1059.15	1001.38	983.50
250000	3507.43	7089.49	**7127.40**	6765.60	5976.79	4643.99
1000000	4100.13	7284.16	**7344.20**	6881.05	5888.30	4605.15
25000000	4892.32	9000.22	**9107.15**	8378.24	7270.69	5657.31
100000000	5042.93	9309.00	**9326.04**	8762.94	7600.75	5868.93

4.2.2 Total program time

Serial baseline

Table 11: Total program time in milliseconds - Serial

	O0
10000	728.38
250000	13703.5
1000000	54505.7
25000000	1.491e+06
100000000	5.420e+06

Speedup: Serial baseline

Table 12: Total program time speedup - Serial with optimizations

	O2	O2+xHost
10000	**2.84**	2.48
250000	4.94	4.94
1000000	**5.06**	5.05
25000000	5.65	**5.66**
100000000	**5.14**	5.13

Table 13: Total program time speedup - OpenMP

World size	Total # of threads							
	2	4	8	16	32	64	128	256
10000	1.18	3.48	4.00	**4.22**	4.16	3.86	3.09	1.98
250000	8.59	17.52	25.30	35.86	38.59	**43.48**	38.56	27.65
1000000	9.73	19.66	29.99	46.90	67.57	**80.22**	72.54	57.39
25000000	11.29	21.81	40.10	68.62	100.68	115.27	129.24	**132.93**
100000000	10.36	20.03	34.32	63.26	94.04	109.99	122.31	**126.73**

Table 14: Total program time speedup - MPI - 1 node

World size	Total # of processes across nodes							
	2	4	8	16	32	64	128	256
10000	0.45	0.46	**0.48**	**0.48**	0.47	0.42	0.37	0.22
250000	4.59	6.27	7.28	7.63	**7.68**	7.00	4.96	2.05
1000000	8.02	12.95	18.99	22.78	**25.24**	22.81	14.96	6.43
25000000	11.18	21.40	41.95	70.04	102.03	**112.96**	110.73	48.40
100000000	10.22	19.65	39.74	68.25	102.01	**113.32**	109.46	90.10

Table 15: Total program time speedup - MPI - 2 nodes

World size	Total # of processes across nodes								
	2	4	8	16	32	64	128	256	512
10000	0.41	**0.43**	**0.43**	0.42	0.38	0.33	0.23	0.14	0.07
250000	4.44	5.70	6.55	**6.78**	6.53	5.43	3.79	2.00	0.88
1000000	7.72	12.04	16.53	20.38	**21.74**	18.12	12.54	6.31	2.43
25000000	10.99	20.50	36.28	61.59	92.19	115.68	**123.74**	59.62	21.10
100000000	10.05	18.88	34.07	61.07	92.76	118.81	**134.52**	109.17	43.04

Table 16: Total program time speedup - MPI - 4 nodes

World size	Total # of processes across nodes									
	2	4	8	16	32	64	128	256	512	1024
10000	0.42	0.39	**0.43**	**0.43**	0.40	0.36	0.28	0.18	0.12	0.03
250000	4.46	5.39	6.13	6.72	**6.77**	6.07	4.45	2.50	1.41	0.59
1000000	8.13	11.30	16.44	18.44	19.08	**19.71**	14.00	8.47	4.24	1.38
25000000	12.59	20.79	36.99	47.14	70.74	96.11	**120.75**	83.88	34.99	11.06
100000000	11.53	19.18	34.91	46.68	74.08	104.07	128.90	**144.76**	63.59	21.18

Table 17: Total program time speedup - MPI - 8 nodes

World size	Total # of processes across nodes									
	2	4	8	16	32	64	128	256	512	1024
10000	0.40	0.36	0.36	**0.41**	**0.41**	0.38	0.32	0.24	0.12	0.08
250000	4.43	5.53	6.09	**6.78**	6.77	6.23	4.97	3.12	1.72	1.12
1000000	7.94	11.84	15.51	20.20	**21.67**	20.46	16.03	10.41	5.27	2.27
25000000	12.41	20.69	33.34	57.33	81.22	106.30	**125.18**	89.17	48.38	16.55
100000000	11.15	19.25	34.27	57.67	88.51	114.81	136.37	**149.34**	91.25	32.85

Table 18: Total program time speedup - Hybrid - 1 node

World size	Total # of processes $\times$ # of threads per process						
	2×128	4×64	8×32	16×16	32×8	64×4	128×2
10000	0.45	0.46	**0.47**	0.46	0.45	0.40	0.31
250000	7.80	**8.12**	8.03	8.04	7.63	6.63	4.21
1000000	25.54	25.83	**25.99**	25.37	24.67	20.81	12.07
25000000	122.30	126.08	117.95	126.57	**129.80**	118.15	106.64
100000000	126.16	128.34	131.51	132.72	**134.60**	130.32	105.64

Table 19: Total program time speedup - Hybrid - 8 nodes

World size	Total # of processes × # of threads per process							
	8×256	16×128	32×64	64×32	128×16	256×8	512×4	1024×2
10000	0.34	**0.38**	0.37	0.36	0.28	0.24	0.12	0.07
250000	6.28	**6.8**	5.94	4.27	4.48	3.18	1.8	1.08
1000000	21.98	**22.34**	20.09	19.72	14.41	11.68	5.51	2.33
25000000	139.46	142.58	141.12	145.01	**145.38**	110.82	53.37	22.02
100000000	139.77	148.43	149.94	162.06	**163.54**	152.58	98.40	38.01

Table 20: Total program time speedup - CUDA

World size	Block size					
	32	64	128	256	512	1024
10000	0.39	**0.46**	0.45	**0.46**	**0.46**	**0.46**
250000	8.51	8.48	**8.55**	8.47	8.51	**8.55**
1000000	32.37	32.63	32.37	**32.83**	32.72	32.74
25000000	488.81	510.12	**513.62**	507.02	503.25	495.32
100000000	745.37	796.48	**799.70**	793.28	785.10	762.86

4.3 Analysis

Whenever any form of parallelism is pursued, it generally relies on a few compelling advantages that are inherent in these methodologies. These advantages can be summerized in three motivations:

- *Speedup*, according to which most programs would run faster if parallelized as opposed to being executed serially. Its main advantage is that it allows a problem to be modelled faster by allowing the concurrent development of multiple execution flows.
- *Accuracy*, according to which more parallel processes would allow for a better model, in terms of diagnostics, approximation error and overall quality of the solution.
- *Scaling*, perhaps the most enticing of the three motivations, according to which more parallel processes would help model a complex problem in the same timespan it would take for fewer processes to model a smaller portion of the very same problem.

It can be noted that in the above lines we've discussed *modelling* a problem, not solving it, and that's because given a model that focuses on those three motivations, many different solutions can be formulated in accordance to a plethora of tools. **In the Game of Life, we opted to focus only on speedup and scaling.** Indeed, being it a zero-player game whose evolution relies on deterministic rules, as we have already long discussed throughout the report, the accuracy constraint is given.

When enforcing speedup and scaling, two main issues should be emphasized, as they pose severe limitations that may clamp parallelism's advantages:

- *Communication overhead*, that is, the time that is lost purely in long awaited communication holds that are needed to take place before and after calculations. During this time, valuable data is being communicated, but no actual progress i being made on completing the algorithm. Such an issue can quickly overwhelm the total time spent solving a problem, sometimes even to the point of making a parallel program less efficient than its serial counterpart.
- *Strong scaling vs Weak scaling*, that is, understanding the impenetrable limitations posed by serial regions to increasing parallelism and how that effect is propagated when the problem complexity scales as well to more and more articulate settings. As we recall, these two concepts are formally summarized by Amdahl's and Gustafson's laws [5].

Taking all of that into consideration, we will now reason on the results we presented in Section 4.2 trying to give an answer to the following question: *Have we achieved acceptable performance on the Game of Life software for a suitable range of data and the HPC resources that we got access to?*

4.3.1 Speedup

In this section, we indicate with T_C and T_P the cumulative generation time and total program time, respectively. From Tables 2-10 in Section 4.2.1, we can make the following observations:

- **Table 2**: the two programs, respectively vectorized with flags *O2* and *O2 + xHost* via the ICC compiler, produce almost identical results for the speedup, with respect to all the possible world sizes. In particular, the improvement in cumulative generation time with respect to the non-optimized program, compiled via the ICC compiler with flag *O0*, has mean 5.3 and standard deviation 0.21, meaning that vectorization can only make performance improvements possible up to some extent, and that they don't depend on the problem size, fact which is again confirmed if we look at the rate of change of the speedup, which does not seem to follow a specific pattern when increasing the world size. From the experimental data, therefore, it appears that the compiler is clever enough to provide always the same degree of optimization when considering vectorization only.
- **Table 3**: this table shows what happens when trying to push to the limits the compute capabilities of one node of the INFN HPC Ocapie cluster, which we recall to correspond to 256 parallel computational units, by means of increasing numbers of OpenMP threads. It can be observed that, in general, a number of OpenMP threads equal to 128 proves to be the most effective on a variety of world sizes, namely 10000, 250000, 1000000, 25000000. However, world size 100000000 finds 256 as its best number of computational units. A world size equal to 100000000, therefore, defines for our experiments the turning point in which the scheduling and communication overhead of OpenMP threads becomes negligible with respect to the cumulative generation time improvement, and makes more useful the action of increasing the number of threads for the node being used. As a complementary observation to these latter facts, we can observe that, in general, the speedup decreases when increasing the number of OpenMP threads for small world sizes, e.g. 10000.
- **Tables 4-7**: these tables show cumulative generation time speedup results obtained when both increasing the number of MPI processes and cluster nodes used for performing computations. We can observe that, when considering a single cluster node, each speedup is generally worse than the corresponding one obtained in Table 3, as the communication overhead in MPI includes message passing, which does not make use of shared memory, whereas by increasing the number of cluster nodes we gradually observe a performance improvement, as the MPI processes become more and more evenly distributed across nodes. In particular, when comparing tables associated with numbers of nodes 1 and 2, the same pattern of best results is maintained, and a relevant the performance improvement regards only pairs $(100000000, 256)$ and $(25000000, 128)$, whereas when considering Table 6-7, the improvement regards only pair $(100000000, 256)$, because the communication latency between nodes, which have been increased, becomes less negligible, and we need more computational units to compensate this fact.
- **Tables 8-9**: in case of experiments performed with the Hybrid MPI-OpenMP architecture, we can observe that, in general, the best speedups are provided by the left-most sections of the two tables, which include, respectively, ranges $[2 \times 128, 8 \times 32]$ and $[8 \times 256, 32 \times 64]$. The motivation for this latter fact is that the workload is balanced between OpenMP threads and MPI processes, and from an intuitive point of view, the best solution is the one in which few MPI processes perform message passing while performing their own computations with the maximum possible useful number of OpenMP threads, where the term "useful" means being big enough to handle the complexity of the problem, but not too much in order to avoid local OpenMP overheads.
- **Table 10**: the speedup values obtained when using the CUDA C program which implements memory coalescing greatly outperform all the other CPU-based implementations. This is expected, as GPUs provide optimal performance when trying to solve problems in which an algorithm must perform simple operations on input data of a grid-like shape, like in this case. In particular, the best value for the CUDA block size appears to be 128 for all the world sizes used in our experiments, and in general, relevant values for the block size appear to be contained inside range $[64, 256]$, whose performance values are similar, unlikely when compared with all the other ones obtained for $32, 512, 1024$. Motivations which complement

these latter observations about the block size range $[64, 256]$ can be found in Section 4.1, when we explain the choice for the CUDA block sizes used in the experiments.

We can perform the same observations when considering the results reported in Section 4.2.2, with the following differences:

- **Table 12**: In this case, since we are using the total program time, the speedup provided by the vectorization only is smaller when considering world size equal to 10000. This may happen because, in that case, the cumulative generation time does not dominate the other execution times, namely the ones used for initialization purposes, memory copying, file reading etc etc., which may not have been vectorized by the compiler. As a complementary observation to this latter fact, all the other speedup values begin to be similar to the ones in Table 2 starting from world size 1000000.
- **All total program time tables**: in general, the speedup obtained when using the total program time metric is lower than the one in the corresponding tables produced for the cumulative generation time. This happens because the total program time includes also initialization operations of various kinds, which may be efficiently parallelized or not, depending on the program implementation being considered (see sections 3 for more informations about the initialization operations and the algorithm design choices). Even if we are mainly interested in the execution time effectively needed to perform the GoL timesteps (cumulative generation time), when using the total program time metric, initialization operations are a huge bottleneck, and this can be experimentally motivated by comparing the best speedup result obtained for the single node OpenMP case and the best one of the 8-nodes Hybrid case, which are respectively 132.93 and 163.54 and hence don't differ too much like in the cumulative generation time case. A complementary observation to this latter one can be made by considering the first row of the 8-nodes Hybrid table, in which, for small world sizes, namely 10000, the speedup becomes a slowdown in performance, as the initialization time surpasses a lot the cumulative generation time. A future improvement on the results obtained with the total program time may therefore be a further optimization of all the operations which are necessary to be executed before starting the first GoL generation, even though, on the other hand, the CUDA program still allowed us to obtain a more significant speedup of 799.70 over the serial baseline.

To complement Tables 2-20 and provide a more insightful visualization of speedups and execution times, we report the line charts contained in Figures 8, 9, 10, 11, 12, 13, 14, 15. In particular, Figure 8 shows, for each world size used for the experiments, the speedup values obtained for an increasing number of OpenMP threads, and confirms again the fact that, on a single node, the amount of computational units used should be appropriate to the size of the problem being solved, as a comparison between the cases of world size equal to 10000 and 100000000 and can testify. The plot in Figure 8 is complemented by the one in Figure 9, in which it is shown how, in general, that the execution time decreases as we increase the number of OpenMP threads. The observations made for Figure 9 do not necessarily hold for plots 10 and 11 when considering MPI processes distributed in nodes:

- Figure 10 confirms again the fact that the overall number of MPI processes should be evenly distributed across the nodes of the cluster, and that ideal values for this parameter can be found in the range $[128, 256]$ when solving the GoL problem.
- Plot 11 completes Figure 10: the communication overhead in MPI message passing becomes a bottleneck starting from an overall number of 512 processes and the execution time reverses its trend.

For what instead concerns the Figures 12 and 13, it is visually confirmed that the best values for the block size are found in the range $[64, 256]$. Moreover, the CUDA-parallelized program proves again the most efficient among the parallel implementations, as Figure 15 testifies: the performance of the best CPU-based solution is far less than the GPU-based one. Despite so, we still reach quite remarkable results by relying on a Hybrid MPI-OpenMP architecture, as when dealing with problem sizes equal to 100000000 the parallelized program is approximately 2000 times faster than the serial implementation.

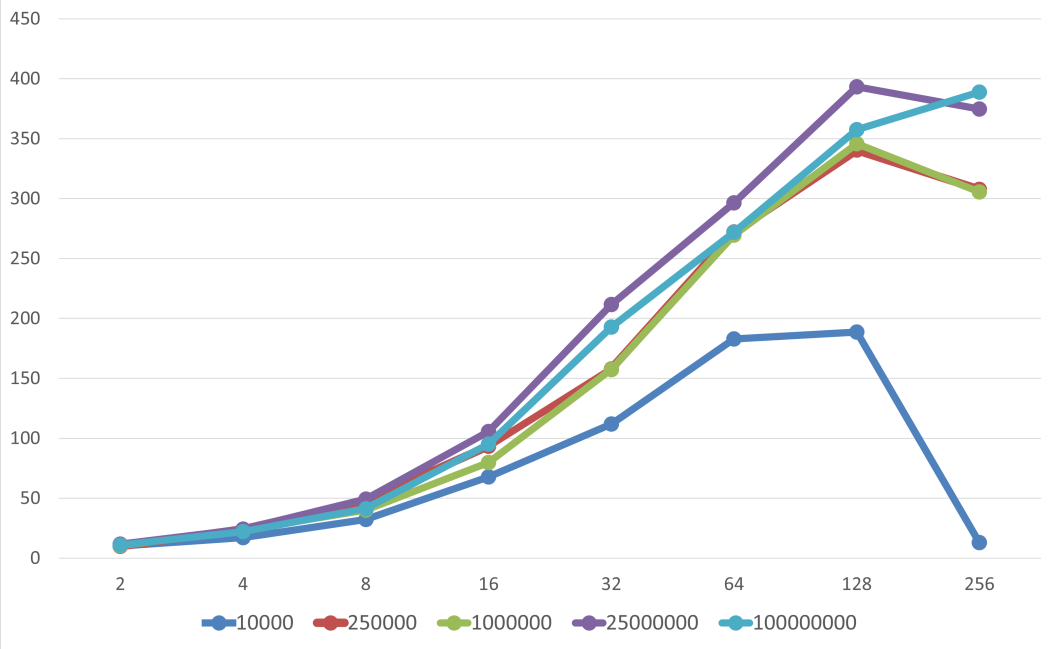


Figure 8: Cumulative generation speedup - OpenMP on different # of cells

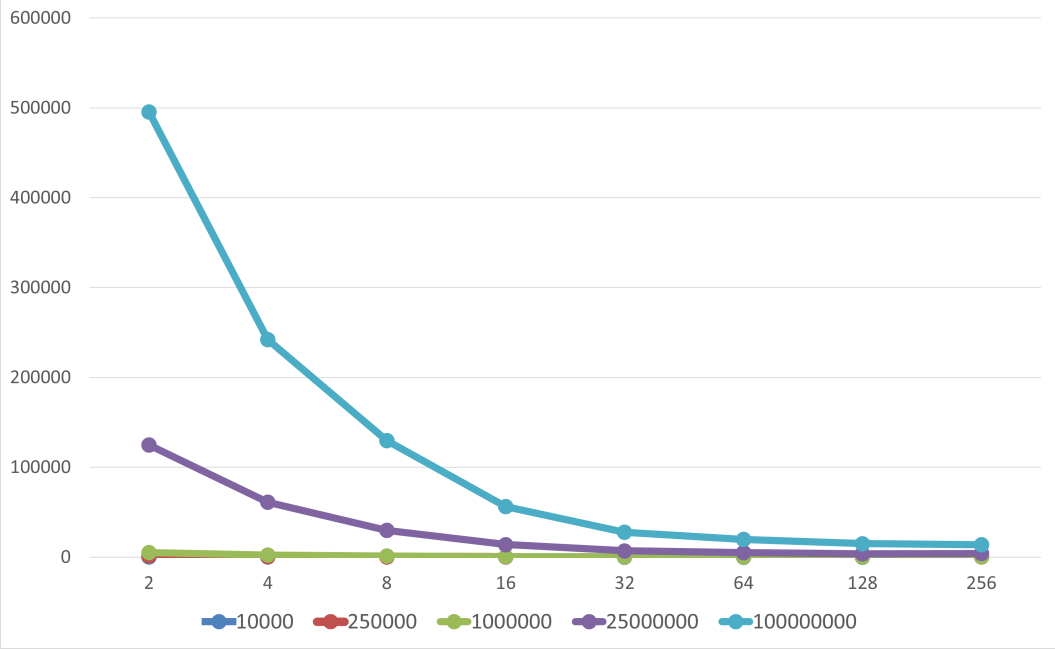


Figure 9: Total program time in milliseconds - OpenMP on different # of cells

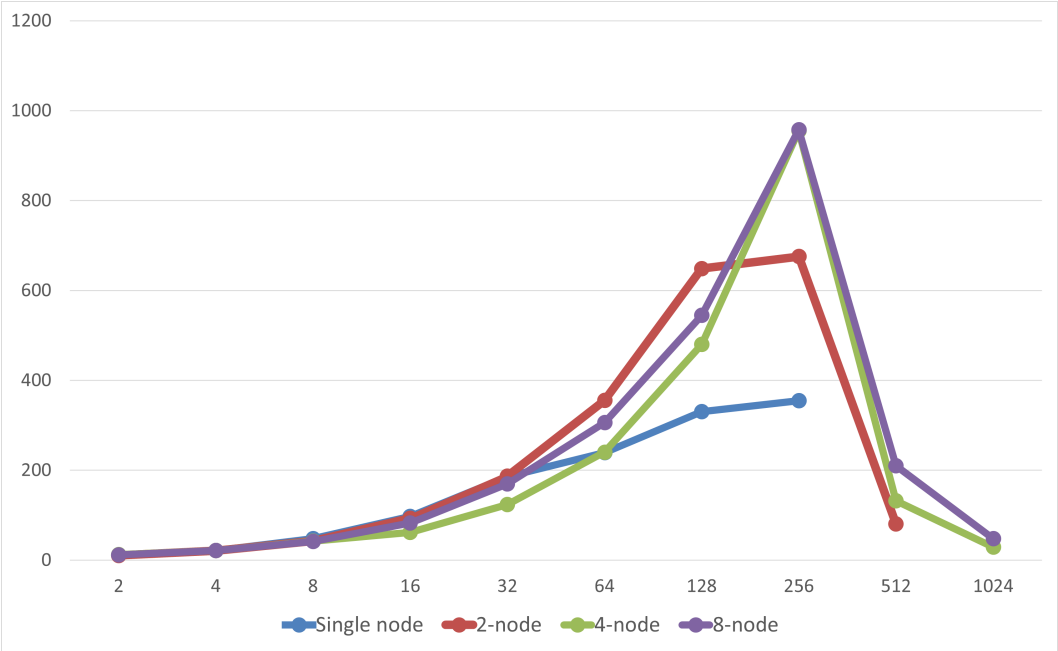


Figure 10: Cumulative generation speedup - MPI on 100000000 cells

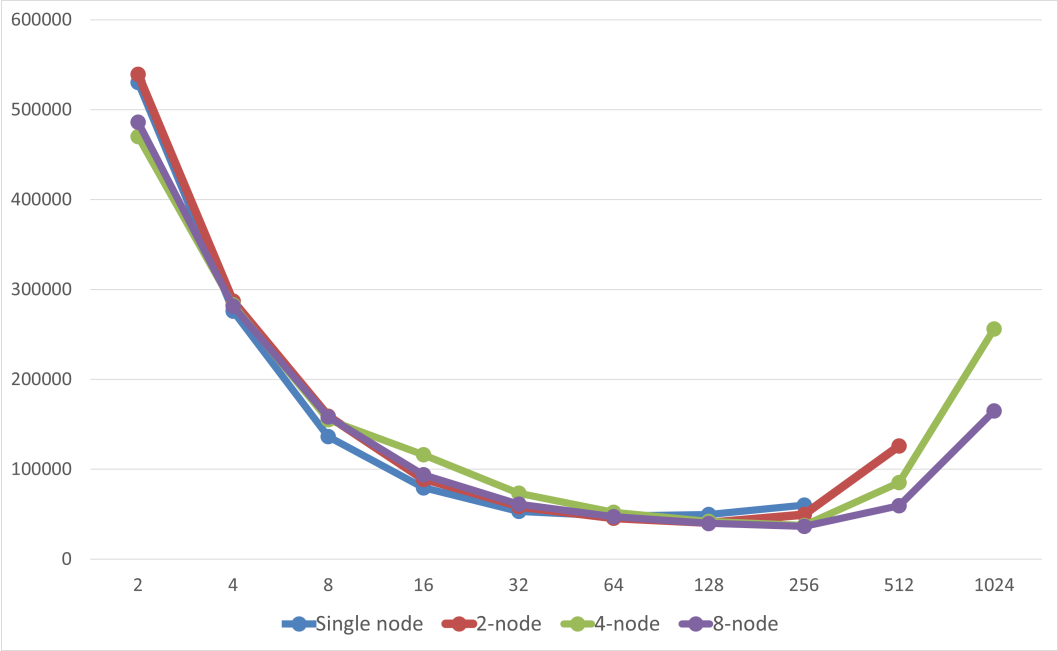


Figure 11: Total program time in milliseconds - MPI on 100000000 cells

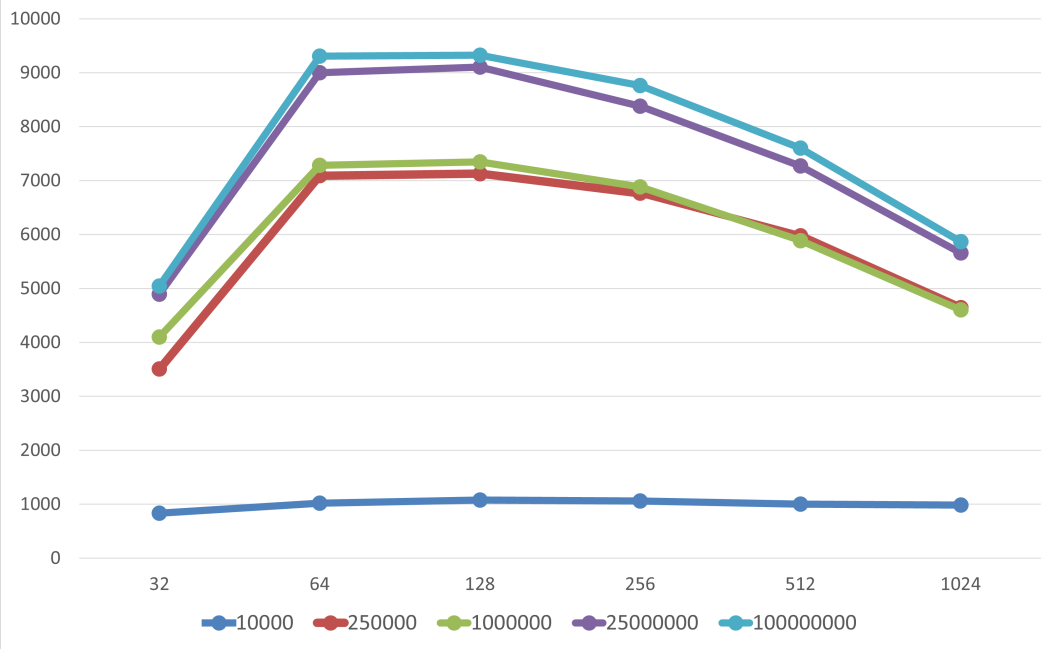


Figure 12: Cumulative generation speedup - CUDA on different # of cells

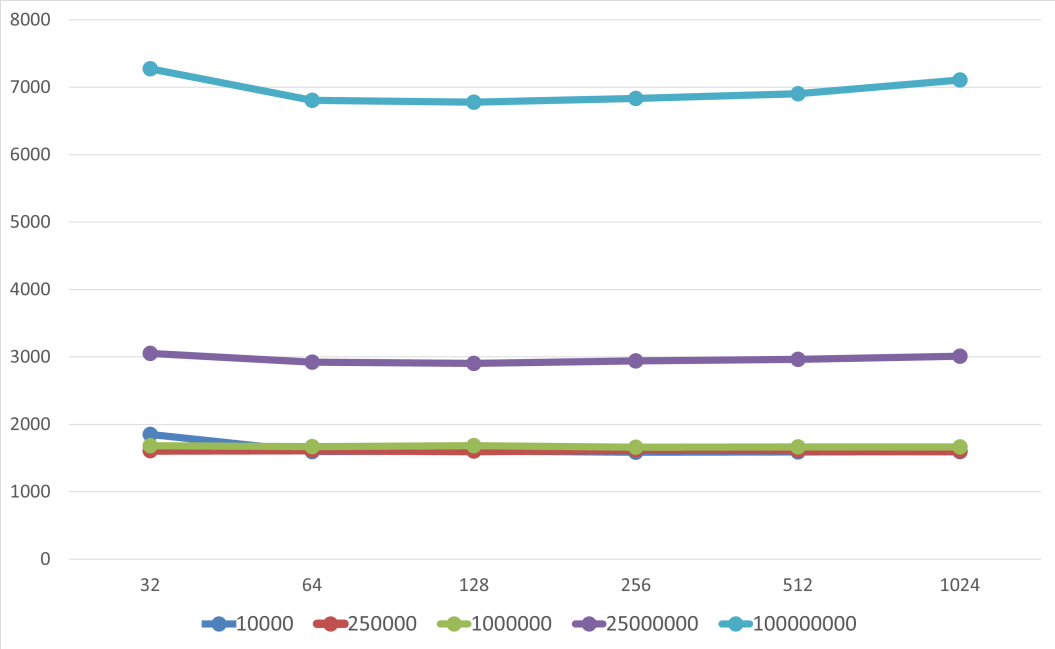


Figure 13: Total program time in milliseconds - CUDA on different # of cells

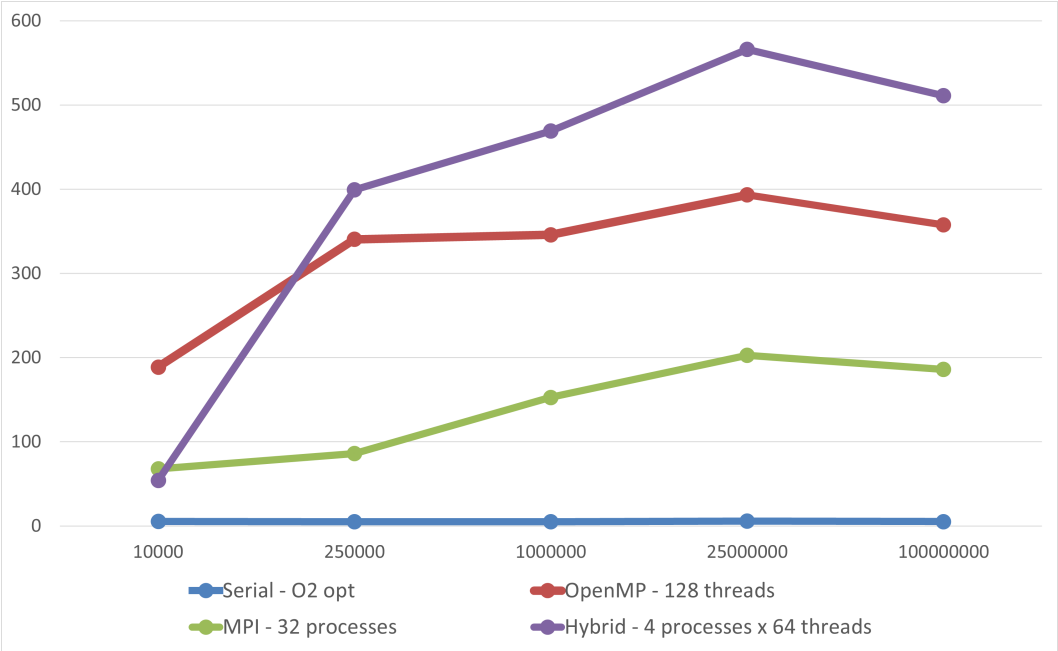


Figure 14: Cumulative generation speedup - Best CPU implementations

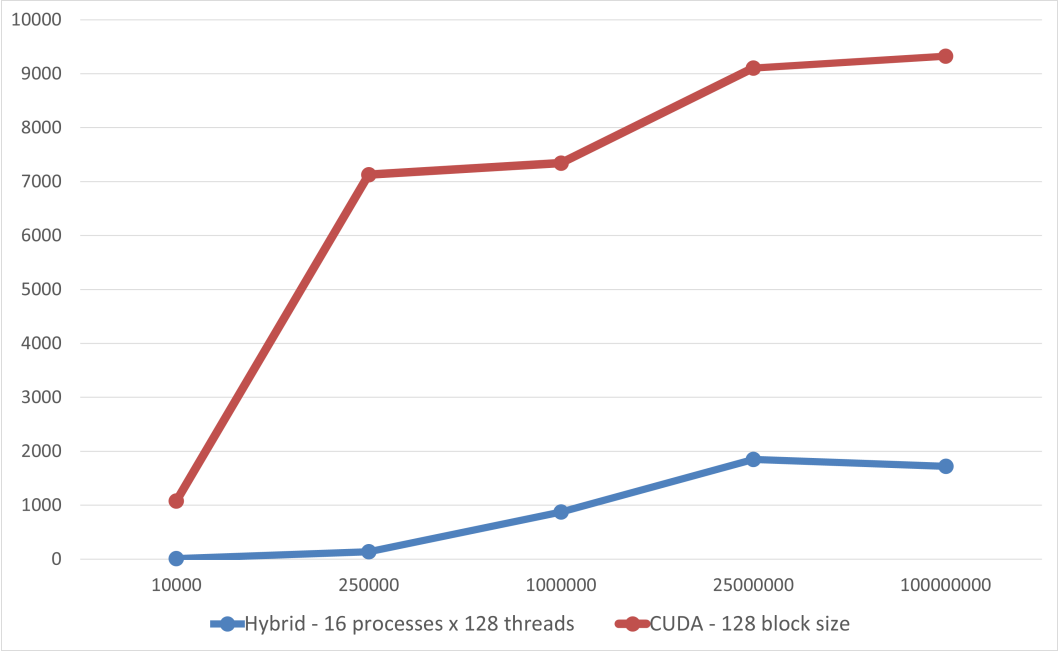


Figure 15: Cumulative generation speedup - Best CPU vs GPU implementations

4.3.2 Scaling

In this section we analyze the obtained result in light of the scaling factor introduced in Section 4.3.

Amdahl's law reveals the limitations of what is know as **strong scaling** in which the number of processes increase the problem size remain constant. The mathematical formulation is the following:

$$Speedup = \frac{1}{1 - P + \frac{P}{N}}, \tag{25}$$

where *P* is the portion of the program that can be parallelized, and *1-P* the portion of the program that remains serial; the *N* represents the number of processors.

Gustafson's law instead states that bigger problems can be modeled in the same amount of time as smaller problems if the processor count is increased. The mathematical formulation of the mentioned law is the following:

$$Speedup(N) = N - (1 - P) * (N - 1), \tag{26}$$

which reveals the promise of **weak scaling**, in which the number of processes increase along with the problem size. In the above formulation *N* represents the number of processors and *1-P* represents the portion of the program that cannot be parallelized.

We now have to verity the above laws in relation with the results that we obtained. Because of the limitations of the Amdahl's law we expect that the algorithm will not scale indefinitely if we only increase the computational capability for computing the same amount of data (strong scaling). This is mainly do to *communication overheads* which will eventually overwhelm the total amount of time spent in running the code. This law is indeed verified in our experiments, and it is visible in Fig. 10. In the plot we can see that as we increase the number of computational units, maintaining fixed the problem size, we have a drop in speedup passed a certain threshold. This is also emphasized in Fig. 11 where it can be seen how the execution time, after dropping considerably as we increase the computation units, increases after passing a certain threshold.

For what concern instead Gustafson's law and weak scaling, we have to verify that if we also increase the problem size when increasing the computational resources, we are able to solve bigger problems in the same amount of time. We also partially verified this law during our experiments. We say partially because we were not able to build a valid setup on which we could maintain a fixed ratio between the computational resources increment and the data increment. Nonetheless if we look for example at Table 27, and we consider the following steps:

- 10000 elements with 2 processes
- 250000 elements with 32 processes
- 400000000 elements with 512 processes,

we can see that there is a speedup improvements considering the problem size and the computational resources as well as a general consistency when it comes down to a comparable data load.

5 Conclusions

We conclude this report by giving a sensible answer to the question previously posed in the end of Section 4.3 in light of the tools that we just observed: *Have we achieved acceptable performance on the Game of Life software for a suitable range of data and the HPC resources that we got access to?*. The answer in our opinion is ***yes***, and we base our statement in light of the results that have been presented in the above plots. In particular we can notice that:

- With OpenMP, in the best case we had a speedup of almost 400 times with respect to the serial execution, which goes on par with the corresponding amount of parallelization, as OpenMP threads are very lightweight and provide great support;
- With MPI, in the best case we had a speedup of almost 1000 times with respect to the serial execution, which gets more and more evident the bigger the ingested the data load is. Indeed not always were they the best choice, and especially, not always the larger bunch of processes was the most suitable choice; but, as we can see from Table 26 and 27, as the computation becomes more demanding, with a suitable range of data load, a bigger amount of processes works better despite the message passing overhead;

- With CUDA, in the best case we had a speedup higher than 9000 times with respect to the serial execution, which was to be expected as GPU processing is tailored with exceptional performances for this kind of repetitive on-grid problems.

Acknowledgments

We would like to acknowledge Prof. Daniele D'Agostino for his availability throughout the development of the project, which helped us reason about its most challenging aspects.

References

[1] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, et al. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[2] B. Gaster et al. Single Instruction Multiple Data.

[3] Design Guide. Cuda c best practices guide. *NVIDIA, July*, 2013.

[4] M. Harris. Efficient global memory accesses.

[5] Anshu Kumari. An Analytical Study of Amdahl's and Gustafson's Law. *SSRN*.

[6] Simon A. Levin. Self-organization and the Emergence of Complexity in Ecological Systems. *BioScience*, 55(12):1075–1079, 12 2005.

[7] Ramin Mafi. *GPU-based Parallel Computing for Nonlinear Finite Element Deformation Analysis*. PhD thesis, 2014.

[8] L. De Mol. An Introduction to Turing Machines.

[9] Stackoverflow. Global memory vs shared memory.

[10] Stackoverflow. How is memory coalescing achieved?

[11] Stackoverflow. Warp divergence discussion.

[12] Wikipedia. Cellular Automation: Guns.

[13] Wikipedia. Cellular Automation: Spaceships.

[14] Wikipedia. Cellular automaton.

[15] Wikipedia. Finite-different Class of Methods.

[16] Wikipedia. Halting Problem.

[17] Wikipedia. John von Neumann's Universal Constructor.

[18] Wikipedia. Moor Neighborhood .

[19] Wikipedia. Simultaneous Multi-threading.

[20] Wikipedia. Turing-completeness.

[21] Wikipedia. Von Neumann Neighborhood.

[22] A. Zucconi. Let's Build a Computer in Conway's Game of Life.

Appendices

A Execution times

In this section we report, for the sake of completeness, the results for the cumulative generation time and total program time metrics which were used to compute the speedups analyzed in Section 4.3. The metrics were measured in milliseconds.

A.1 Cumulative generation time

Table 21: Cumulative generation time in milliseconds - Serial

	O0
10000	550.76
250000	13328.2
1000000	53465.8
25000000	1.471e+06
100000000	5.351e+06

Table 22: Cumulative generation time in milliseconds - Serial with optimizations

	O2	O2+xHost
10000	104.41	104.41
250000	2585.09	2585.85
1000000	10365.3	10364.7
25000000	257900	257647
100000000	1.0316e+06	1.03222e+06

Table 23: Cumulative generation time in milliseconds - OpenMP

World size	Total # of threads							
	2	4	8	16	32	64	128	256
10000	54.72	32.2	17	8.11	4.92	3.01	2.92	42.69
250000	1338.56	553.83	279.61	142.69	84.37	48.98	39.15	43.34
1000000	5099.65	2273.04	1335.89	670.73	339.59	198.36	154.58	174.96
25000000	124637	61025.7	29904.9	13917.8	6955.89	4964.14	3743.4	3927.04
100000000	495359	242156	129579	56207.3	27741.4	19679.8	14970.7	13766
400000000	1.97889e+06	975261	479665	222621	110630	79514.3	59705.1	48501.8
1600000000	8.27462e+06	4.11123e+06	2.02099e+06	899156	447937	371832	268298	186202

Table 24: Cumulative generation time in milliseconds - MPI - 1 node

World size	Total # of processes across nodes							
	2	4	8	16	32	64	128	256
10000	54.5	27.36	16.88	9.81	8.14	34.64	142.2	386.28
250000	1259.44	608.32	292.45	154.43	155.16	260.1	821.62	3124.7
1000000	4994.03	2407.09	1097.88	614.12	350.17	481.01	1533.33	5235.18
25000000	125570	61904.3	28090.8	14027.4	7260.1	5600.07	4048.4	17549
100000000	502820	249007	111739	55154.2	28782.4	22388.3	16187.8	15072.8
400000000	2.00838e+06	987717	450141	219517	111685	89287.8	64885	54153.8
1600000000	8.03344e+06	3.96101e+06	1.79666e+06	876627	444476	323572	243809	214972

Table 25: Cumulative generation time in milliseconds - MPI - 2 nodes

World size	Total # of processes across nodes								
	2	4	8	16	32	64	128	256	512
10000	53.76	29.96	17.27	11.16	8.69	35.79	92.01	155.59	301.08
250000	1269.46	637.35	314.62	156.74	157.09	262.32	526.18	1732.56	5624.34
1000000	5073.85	2541.02	1268.98	629.29	342.86	484.69	974.53	3292.89	11020.8
25000000	126155	63202.6	31554.6	14993.3	7166.5	3770.68	2071.16	10819	46375.8
100000000	503914	252782	125831	57344.3	28617.4	15057	8248.5	7922.34	66564.4
400000000	2.01496e+06	1.00816e+06	503212	229528	109295	60183.9	33232.1	31331.3	34280.2
1600000000	8.04895e+06	4.02553e+06	2.00302e+06	905478	439516	220128	131726	125349	137786

Table 26: Cumulative generation time in milliseconds - MPI - 4 nodes

World size	Total # of processes across nodes									
	2	4	8	16	32	64	128	256	512	1024
10000	46.68	27.79	21.38	13.04	9.28	35.6	91.85	92.74	168.92	1708.48
250000	1094.62	634.9	464.53	225.96	156.82	262.03	524.99	1079.81	3526.99	6540.55
1000000	4377.9	2524.63	1267.09	893.33	448.3	482.78	986.05	2116.32	6895.47	23195
25000000	109386	62822.7	31505	21772.6	11139.3	5569.16	2796.18	6801.67	28381.2	104605
100000000	437268	252078	125742	86556.5	43314.5	22267.6	11146.8	5602.37	40607.4	182408
400000000	1.74767e+06	1.00594e+06	502862	337940	174861	88875.9	44455	22291.7	20261.3	258382
1600000000	7.94321e+06	6.86778e+06	2.12276e+06	1.00234e+06	453801	218825	177663	89181.2	80837.7	97684

Table 27: Cumulative generation time in milliseconds - MPI - 8 nodes

World size	Total # of processes across nodes									
	2	4	8	16	32	64	128	256	512	1024
10000	46.28	27.97	17.84	13.27	10.06	40.73	92.6	98.65	95.74	173.66
250000	1298.14	617.31	321.98	186.57	157.63	262.41	524.35	1077.33	2197.5	3536.55
1000000	4847.48	2523.12	1269.4	641.99	380.18	485.22	972.56	2058.39	4296.42	14084.1
25000000	111511	63310.5	35698.6	17279.4	9577.16	5175.58	2788.78	6791.92	17633.9	63505.1
100000000	456633	252149	128655	64563.8	31497.4	17489.5	9820.2	5590.07	25467.8	111416
400000000	1.74969e+06	1.00937e+06	506906	252564	125862	60484.6	40407.6	22265	12461.6	158602
1600000000	6.99391e+06	4.02398e+06	2.01781e+06	1.02428e+06	503108	279034	160813	88427.4	49916.7	58468.8

Table 28: Cumulative generation time in milliseconds - CUDA

World size	Block size					
	32	64	128	256	512	1024
10000	0.66	0.54	0.51	0.52	0.55	0.56
250000	3.8	1.88	1.87	1.97	2.23	2.87
1000000	13.04	7.34	7.28	7.77	9.08	11.61
25000000	300.84	163.53	161.61	175.67	202.43	260.16
100000000	1061.09	574.82	573.77	610.64	704.01	911.75
400000000	4020.46	2197.77	2181.89	2319.33	2682.22	3532.27
1600000000	16103.3	8798.76	8743.66	9293.69	10726.3	14089.9

A.2 Total program time

Table 29: Total program time in milliseconds - Serial

	O0
10000	728.38
250000	13703.5
1000000	54505.7
25000000	1.491e+06
100000000	5.420e+06

Table 30: Total program time in milliseconds - Serial with optimizations

	O2	O2+xHost
10000	256.37	294.05
250000	2772.13	2775.64
1000000	10764.5	10790.8
25000000	263869	263548
100000000	1.05474e+06	1.05567e+06

Table 31: Total program time in milliseconds - OpenMP

World size	Total # of threads across nodes							
	2	4	8	16	32	64	128	256
10000	619.44	209.03	182.03	172.55	175.12	188.52	235.79	367.35
250000	1595.63	782.11	541.69	382.1	355.14	315.14	355.39	495.6
1000000	5602.68	2772.71	1817.17	1162.25	806.7	679.44	751.35	949.72
25000000	132101	68365	37178.1	21727.6	14809.1	12935.3	11536.8	11216.5
100000000	523203	270680	157950	85691.5	57641.7	49285.1	44320.2	42774.6
400000000	2.09068e+06	1.088e+06	591542	335132	224224	192647	173043	162498
1600000000	8.73332e+06	4.56658e+06	2.47273e+06	1.35357e+06	903315	825243	722392	640618

Table 32: Total program time in milliseconds - MPI - 1 node

World size	Total # of processes across nodes							
	2	4	8	16	32	64	128	256
10000	1601.5	1577.53	1523.17	1507.04	1558.87	1739.82	1958.73	3279.91
250000	2983.66	2187.1	1883.24	1795.94	1783.87	1957.59	2762.8	6682.65
1000000	6792.08	4207.84	2869.63	2392.41	2159.6	2389.61	3644.48	8474.08
25000000	133349	69686.4	35541.3	21289.3	14613.9	13199.3	13464.9	30804
100000000	530140	275919	136419	79420.4	53138.3	47835.4	49522.5	60164.6
400000000	2.11312e+06	1.09095e+06	544934	311963	204454	189879	192375	227316
1600000000	8.44011e+06	4.36609e+06	2.16224e+06	1.23416e+06	804739	699064	734519	903751

Table 33: Total program time in milliseconds - MPI - 2 nodes

World size	Total # of processes across nodes								
	2	4	8	16	32	64	128	256	512
10000	1786.33	1698.63	1687.4	1727.7	1928.4	2227.82	3207.4	5282.49	10307.7
250000	3083.03	2403.18	2091.73	2020.74	2098.14	2525.77	3614.02	6837.61	15511.6
1000000	7063.6	4526.15	3297.95	2674.38	2507.2	3008.7	4345.34	8631.92	22472.1
25000000	135620	72741.3	41102.4	24208.9	16172.7	12889.2	12049.3	25010.4	70670.3
100000000	539346	287054	159125	88765.3	58435.2	45624.2	40295.5	49655.7	125941
400000000	2.14235e+06	1.13733e+06	630098	346242	223361	173955	148045	178248	241318
1600000000	8.55035e+06	4.52131e+06	2.50037e+06	1.35766e+06	876534	659169	569410	702396	935096

Table 34: Total program time in milliseconds - MPI - 4 nodes

World size	Total # of processes across nodes									
	2	4	8	16	32	64	128	256	512	1024
10000	1716.14	1871.44	1677.84	1695.45	1807.76	2017.23	2593.57	3985.5	6215.44	26796.4
250000	3072.45	2541.59	2237.14	2038.13	2024.66	2258.34	3081.56	5487.98	9708.33	23310.9
1000000	6707.81	4825.26	3314.98	2956.45	2857.03	2765.62	3892.02	6433.32	12851.4	39580.9
25000000	118446	71709.2	40308.2	31631.9	21076.2	15513.4	12348.5	17774.8	42613	134860
100000000	470119	282689	155265	116124	73172.8	52089.1	42054.9	37444.8	85238.3	255993
400000000	1.86113e+06	1.12071e+06	614760	451090	287715	201665	157297	137796	171495	483012
1600000000	8.38654e+06	7.31161e+06	2.56425e+06	1.44183e+06	895399	655927	616152	527260	652319	917545

Table 35: Total program time in milliseconds - MPI - 8 nodes

World size	Total # of processes across nodes									
	2	4	8	16	32	64	128	256	512	1024
10000	1813.15	2036.45	2050.71	1763.7	1790.75	1916.45	2255.63	2976.51	5891.94	8789
250000	3093.62	2476.21	2248.83	2022.33	2022.77	2199.32	2757.83	4391.98	7989.75	12191.1
1000000	6861.81	4605.18	3513.14	2698.56	2514.82	2663.69	3399.57	5236.2	10343	24013.4
25000000	120162	72077.1	44726.9	26009.9	18358.2	14027.2	11911.2	16720.7	30821.8	90064.9
100000000	486118	281596	158157	93998.2	61242.8	47215.3	39751.1	36297.6	59407.7	165005
400000000	1.86153e+06	1.12145e+06	619010	364638	237716	171992	152583	135216	130029	309484
1600000000	7.43162e+06	4.46547e+06	2.45763e+06	1.46337e+06	943938	717025	599836	526985	503525	639457

Table 36: Total program time in milliseconds - CUDA

World size	Block size					
	32	64	128	256	512	1024
10000	1849.31	1595.52	1605.77	1584.56	1589.39	1592.86
250000	1610.52	1615.54	1603.17	1618.54	1610.3	1603.22
1000000	1683.69	1670.38	1683.95	1660.48	1665.88	1664.6
25000000	3050.29	2922.89	2902.94	2940.75	2962.77	3010.2
100000000	7272.45	6805.75	6778.4	6833.22	6904.45	7105.71
400000000	23979.4	22228.3	22193.2	22326.9	22715	23562.3
1600000000	91367.6	84611.2	84314.3	85046.3	86182.4	95553.3