Authors: Manuel Parmiggiani & Giacomo Garbarino

High Performance Computing Final Project: Conway's Game of Life

This paper will discuss and compare 4 different implementations of the Conway's Game of Life in order to state which methodology suits better this kind of problem, the behavior of each implementation can be sequential or parallel depending of the employed libraries.

First of all, it's necessary to briefly introduce the rules of this game: the game world is a two-dimensional grid and the smallest unit of this grid will be addressed as a “cell”, at the beginning of the game each cell can be active (ON) or not (OFF) depending of some random initialization, the game will run for a certain number of iterations and at each iteration each cell will be updated according to the following rules:

- If a cell is ON and has fewer than two neighbors that are ON, it turns OFF;
- If a cell is ON and has either two or three neighbors that are ON, it remains ON;
- If a cell is ON and has more than three neighbors that are ON, it turns OFF;
- If a cell is OFF and has exactly three neighbors that are ON, it turns ON.

Each of the presented approaches has its own implementation with the following common features:

- All of the parameters are initialized with default values, some of them (for example, the dimension of the grid) can be replaced with values injected by the user when launching the program;
- The grid of the chosen dimension is created and its cells are randomly initialized with 1 (ON) or 0 (OFF);
- The program counts and prints the number of cells which are active before the beginning of the iterations, then the game evolution starts;
- During each iteration, the evolution of the game is computed (the implementation of this step is what distinguishes one methodology from another) and the time elapsed between the start and the end of that single iteration is calculated and added to a variable representing the total time;
- At the end of the execution, the program counts and prints the number of cells which are still active, then the statistics of the run are saved in a file, together with the total elapsed time.

Since the aim of the experiment was to compare the performance of the different approaches, all of the other features were developed just at the essential level to make all working correctly (this is the reason why neither of the implementations has a GUI or something else which is not fundamental to the performance measurements).

Also the random seed used to generate the grid is kept the same during all the implementations and executions, this was done on purpose since in this way it has been possible to test all of the different implementations against the same grid when dimensions are unchanged.

1 Sequential Implementation

The first and the simplest of the 4 methodologies, in this implementation the grid is simply scanned and updated sequentially, the update is done by defining a new grid matrix for storing the updated values and the effective update will be done at the end of the iteration by swapping the references of the old and the new matrix (since it's the optimal way to avoid computational overheads due to the operation), the same logic will be kept also for the other implementations.

The program takes as an injected input just the size of the grid, passed as a parameter when launching the program from the command line, the program was tested with the following dimensions: 100x100, 500x500, 1000x1000, 5000x5000.

At this point, after all the four executions of this implementation, the results corresponding to the different grid sizes are stored in a csv file;
Now, since the aim of this experiment is to compare different approaches, there is the need to obtain the result of at least another methodology in order to start inferring conclusions.

The first explored alternative methodology is again a sequential implementation which makes use of an additional feature: vectorization.
Vectorization can be done both manually by modifying the code by writing vector instructions and automatically by the compiler, in this case the automatic vectorization of the ICC compiler was employed for the experiment.

In order to register results from both the non-optimized version and the vectorized one, the same source code was compiled respectively with the -O0 and -O2 options, the first option disables all of the automatic optimizations the compiler would have done while the second one enables them.

The difference in performance is computed in % in terms of saved execution time, in this case an optimization of 35% means that the vectorized version of the program saves up 35% of the execution time of the non-optimized version.

Grid Dimension	Non-Optimized (O0)	Vectorized (O2)	Optimization
100 x 100	120.70 ms	77.84 ms	35.49 %
500 x 500	3016.35 ms	1954.90 ms	35.19 %
1000 x 1000	12056.08 ms	7790.69 ms	35.38 %
5000 x 5000	300588.92 ms	194305.47 ms	35.36 %

From the results above, the following conclusions can be inferred:

- The vectorized version is (obviously) faster than the non-optimized one;
- The gained optimization isn't proportional with the problem size since even with a significant difference in the grid dimension the percentage of optimization obtained is almost the same.
- Extra step: trying with O1 optimization the computational time is lower than the non-optimized O0 version but higher than the O2 version while O3 optimization outperforms all of the others.

The computational time used for comparing the sequential implementation with the following ones will be the O0 time.

2 OpenMP Implementation

This second implementation will make use of OpenMP, the difference in the code is minimal but the performance gain will be huge.

By adding the omp.h library, the program now allows parallelism and ad-hoc instructions for a multi-threaded execution;
When running the program, the number of threads will be injected as an extra parameter together with the dimension of the grid.

The most important change with respect to the previous implementation is the addition of a special tag which allows to parallelize the execution of the for loop it was put over, thanks to this the computation of that loop will be divided among a certain number of threads (passed as a parameter while launching the program).

In order to have an optimal comparison, this second program was tested with the same dimensions of the grids used to test the previous one; for each grid, the program was tested with different number of threads in order to observe whether the gained optimization increases of decreases by changing the number of threads.

The following table has a row for each grid size and a column for each number of threads employed, each cell contains the measured optimization with respect to the time saved from the sequential execution against a grid of the same dimension for that grid size and that number of threads:

Grid	5	10	20	30	50	100	150	200
100 x 100	61.04 %	63.64 %	61.54 %	58.20 %	52.59 %	31.40 %	18.40 %	8.45 %
500 x 500	87.21 %	93.23 %	95.72 %	96.47 %	96.89 %	96.43 %	95.21 %	94.45 %
1000 x 1000	88.40 %	94.22 %	96.98 %	97.79 %	98.50 %	97.66 %	98.08 %	98.03 %
5000 x 5000	87.50 %	93.83 %	97.26 %	98.16 %	98.88 %	98.91 %	98.92 %	99.05 %

Since all the percentages are positive, it is possible to state that the OpenMP implementation is better than the O0 sequential implementation.

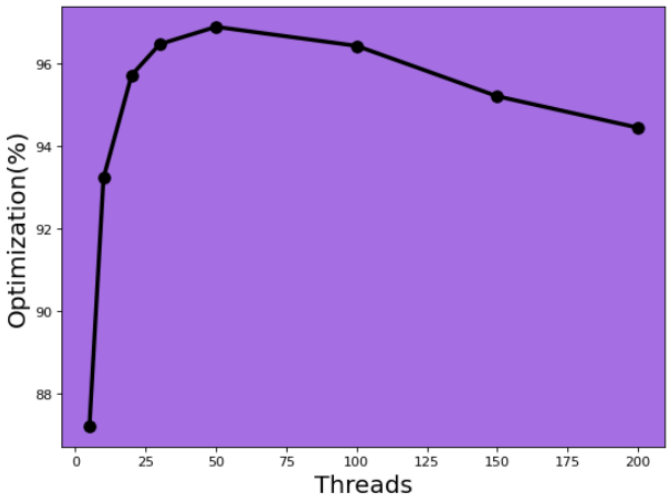
However, by increasing too much the number of threads there is the risk of having too much computational complexity for a too simple problem: that's, for example, the case of the 100x100 grid in which the execution has a significant drop of performance when the number of threads reaches high numbers;

Although being still better than the O0 sequential implementation, the same it's not true for the comparison to the O2 vectorized implementation which saves up a constant 35% of computational time independently from the grid size (outperforming the execution of OpenMP for 100, 150 and 200 threads with the 100x100 grid), this means that for every possible too small size of the grid there may be a theshold number of threads that may establish the boundary between when the OpenMP implementation is preferable to the O2 vectorized implementation and when not.

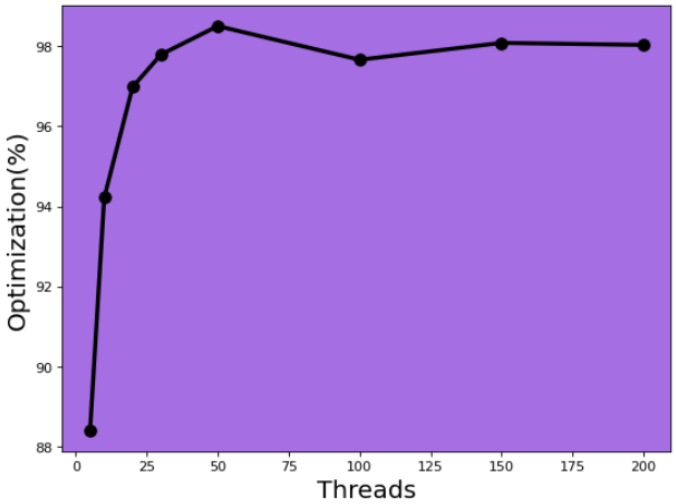
Despite the just evidenced flaw of the OpenMP implementation, it is possible to state that OpenMP is always preferable because even for a small grid there is a certain number of threads that make the execution faster than both the O0 amd the O2 implementations.

Another interesting feature can be observed by plotting some of the numbers of the table above, in particular the ones related to bigger grids: by putting the number of threads on the X axis and the obtained optimizations on the Y axis it is possible to observe how the more the grid size gets bigger the more the obtained curve seems to have an asymptotic behavior with the 100% optimization as an asymptote.

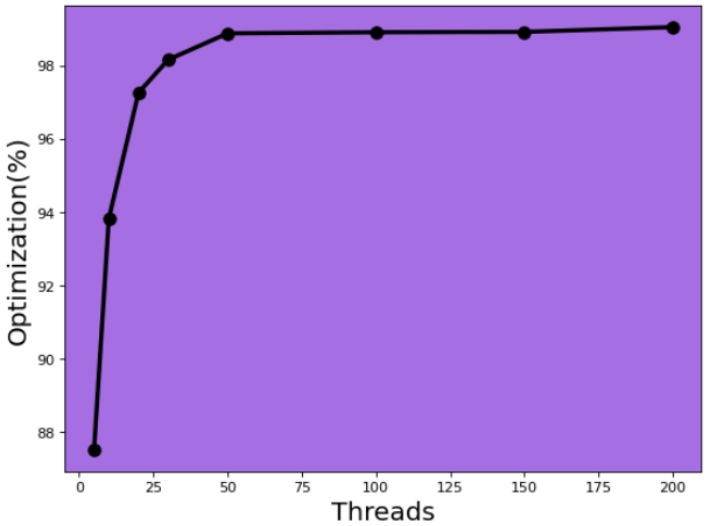
[500 x 500 grid]



[1000 x 1000 grid]



[5000 x 5000 grid]



3 MPI Implementation

This third implementation will make use of the Message Passing Interface (MPI) library, this time the difference in the code will be significant with respect to the previous implementations.

The main difference from OpenMP is that MPI makes use of multiple nodes without the possibility of keeping the data in a shared memory, the difficulty comes from how to ensure that each node receives enough information to correctly continue its part of the work even without having access to all of the grid.
One possible solution that makes use of the message-passing feature of MPI (which allows the nodes to send and receive messages during the execution) can be setting each node to send to the nodes processing the immediately upper and the immediately lower chunk of the grid the rows they need to correctly calculate the new status of the cells in the first and the last rows of their own chunk; this is done by importing the mpi.h library and defining a new MPI_Datatype referring to the rows each node has to send or receive.

For what concerns the execution, the code was tested either by changing the number of nodes than by changing the total number of processes (and so also the number of processes per node); in the cluster in which the code was tested, there were 8 available nodes so the testings were conducted by using 2,4,6 and 8 nodes, the number of processes has to be a multiple of the number of nodes since the load has to be distribuited equally to the nodes, the minimum common mulitple among 2,4,6 and 8 is 24 so the program was tested with 24 processes and higher number of processes that are multiples of 24.

Since there are a lot of results to show and discuss, here there is a table for each grid size, each column of the table corresponds to a total number of processes and each row to the number of nodes, each cell of the table contains the measured optimization with respect to the time saved from the sequential execution against a grid of the same dimension for that number of nodes and processes:

100 x 100 grid

Nodes	24 processes	48 processes	96 processes	192 processes
2	89.43 %	92.68 %	91.75 %	41.46 %
4	89.47 %	92.16 %	91.26 %	58.02 %
6	87.15 %	90.00 %	90.80 %	62.01 %
8	47.41 %	87.40 %	87.18 %	62.99 %

500 x 500 grid

Nodes	24 processes	48 processes	96 processes	192 processes
2	97.25 %	97.90 %	98.12 %	87.32 %
4	97.34 %	97.82 %	98.17 %	87.15 %
6	97.21 %	97.93 %	98.30 %	92.92 %
8	97.16 %	97.70 %	98.02 %	92.71 %

1000 x 1000 grid

Nodes	24 processes	48 processes	96 processes	192 processes
2	98.22 %	98.07 %	98.45 %	97.47 %
4	98.18 %	98.13 %	98.48 %	98.57 %
6	98.10 %	98.16 %	98.48 %	98.64 %
8	97.88 %	98.09 %	98.43 %	98.57 %

5000 x 5000 grid

Nodes	24 processes	48 processes	96 processes	192 processes
2	98.69 %	98.32 %	99.40 %	99.56 %
4	98.52 %	99.32 %	99.63 %	99.70 %
6	98.48 %	98.96 %	99.09 %	99.79 %
8	98.51 %	99.30 %	99.62 %	98.68 %

The just presented results clearly decrees MPI as superior to all of the previously

presented approaches, even in this case the rule “too complex solution over a too simple problem” is true in the case of the 100x100 grid, especially in this case it is possible to observe how:

- too few processes doing too simple tasks over too many nodes may lead to a decrease of the gained performance, this is the case of the highlighted cell in the 100x100 grid's table, maybe this happens because of the overhead due to the cost of exchanging messages between nodes;
- too many processes per node doing too simple tasks may lead to a decrease of the gained performance, this is the case of the last column of the 100x100 grid's table and the reasons are again related to the message exchanging overhead.

CUDA Implementation

As the last possible solution, Compute Unified Device Architecture (CUDA) provides the possibility of employing a GPU in order to accelerate the computation maybe even more than with the other presented methodologies.

From the code perspective, the main difference is the usage of the various CUDA functions for deploying the grid on the GPU (through a deep copy of it) and for parallelize the execution exploiting the additional computational power given by the GPU.

In order to execute the CUDA implementation, there is also the need of switching to a CUDA cluster with respect to the INFN cluster used for all the previous executions, CUDA has also its own compiler and the code has to be deployed over a specific node of the cluster before having the possibility of run it.

The program takes as injected parameters the size of the grid and the number of threads, there is a specific rule to follow in order to decide how many threads run: the GPU multiprocessor schedules threads in groups of 32 threads so the number of threads has to be a multiple of 32 and cannot exceed 1024 which is the maximum number of runnable threads.

The following table presents the results obtained from the execution of the CUDA implementation, each column of the table corresponds to the total number of threads and each row to the grid dimension, each cell of the table contains the measured optimization with respect to the time saved from the sequential execution against a grid of the same dimension for that number of threads:

Grid	32	64	128	256	512	1024
100 x 100	99.88 %	99.90 %	99.90 %	99.90 %	99.90 %	99.89 %
500 x 500	99.97 %	99.98 %	99.98 %	99.98 %	99.98 %	99.97 %
1000 x 1000	99.98 %	99.98 %	99.98 %	99.98 %	99.98 %	99.98 %
5000 x 5000	99.98 %	99.99 %	99.99 %	99.99 %	99.98 %	99.98 %

The results are definetly the best ones from all the implementations, in every situation the optimization is so high that is almost 100% (for obvious reasons it cant't be 100% since for saving up the 100% of the execution time the run has to happen in no time and this is physically impossible), this clearly highlights the computational gain of using a GPU and demonstrates that CUDA is the best among all of the considered alternatives.