Politecnico di Torino

Parallelization of Conway's Game of Life Using CUDA

High Performance Computing

Mirko Ciardo, Riccardo Ghianni, Lorenzo Lucia

Master degree in Quantum Engineering

January 23, 2025

Contents

Abstract

Conway's Game of Life is a cellular automaton that simulates the life and death of cells on a two-dimensional grid based on simple rules. This report presents a parallel implementation of the Game of Life using various approaches: NVIDIA's CUDA platform to harness the computational power of GPUs, OpenMP to perform parallelization without specifical hardware and MPI used in multi nodes systems, some examples of hybrid solutions are also presented. The implementations handle various and large sizes of grids and generations. This document analyzes the problem, details the solution strategy, and examines each part of the program, providing insights into parallel computing techniques applied to cellular automata.

1 Introduction

Conway's Game of Life, devised by John Conway in 1970, is a zero-player game that simulates cellular evolution. Each cell on a grid can be in one of two states: alive or dead. The state of each cell changes over discrete time steps based on the states of its eight neighbors. This simple set of rules can lead to remarkably complex behavior, making the Game of Life a subject of interest in various fields, including mathematics, computer science, and physics.

Simulating large grids over many generations can be computationally intensive due to the sheer number of calculations required. Parallel computing offers a solution by distributing computations across multiple processing units.

2 Background

2.1 Conway's Game of Life Rules

The Game of Life operates on a two-dimensional grid of cells. The evolution of the grid is determined by the following rules applied to each cell:

- **Birth**: A dead cell with exactly three live neighbors becomes alive.
- **Death by Overcrowding**: A live cell with four or more live neighbors dies.
- **Death by Exposure**: A live cell with one or no live neighbors dies.
- **Survival**: A live cell with two or three live neighbors remains alive.

For each initial grid, final state, total alive generations and consecutive alive generations are recorded.

2.2 Need for Parallelization

Simulating the Game of Life on large grids and over many generations requires significant computational resources. Each cell's state depends on its neighbors, making the problem inherently parallelizable. Using parallel computing techniques, we can perform simultaneous updates on multiple cells, greatly reducing execution time.

Methodologies

3 CUDA and GPU Computing

3.1 Why CUDA

CUDA has been chosen as the parallelization technique to solve the Conway Game of Life problem because of one main reason: it supports many (hundreds of thousands) non-communicating parallel threads. In fact, each process must compute the next state of a single cell, the only necessary communication between threads is between the update of the grid to the current value of the cells and the computation of the next turn state.
Other techniques like MPI or OpenMP do not allow for that many parallel processes, therefore one would have had to find a way to conveniently distribute a set of cells to each process.

3.2 Parallelization Strategy

The strategy involves mapping each cell in the grid to a CUDA thread. Each thread computes the next value of its corresponding cell based on its current state and that of the neighbouring cells. Therefore, at each iteration, the kernel should: first update the whole grid with the new cells and then calculate the new state of each cell.
To do so, it is necessary to synchronize every thread after the table has been updated. This is possible using the library *cooperative_groups.h* which allows creating groups of processes (even the whole CUDA grid can be a group) and synchronize them. The latter library is not supported by the GPU mounted on the Hactar cluster, therefore another solution must be implemented.
To achieve coherence between threads at every new iteration in the Conway evolution the program must return to the CPU, to make sure that all threads completed the process, and then launch another kernel. Here, we show the section exploited for the kernel launch:

```
for (int i=0; i < iter; i++) {
    game_iterations<<<gridSize , blockSize>>>(
        dev_mat,
        dev_streak,
        dev_counter,
```

```
        prev,
        n
    );
    int *temp_matrix = dev_mat;
    dev_mat = prev;
    prev = temp_matrix;
}
```

at each iteration the kernel is launched, afterwards, the objects containing the grid and a reference grid are updated.
This technique is slower with respect to the one using *cooperative_groups.h*, but is still capable of achieving impressive results.

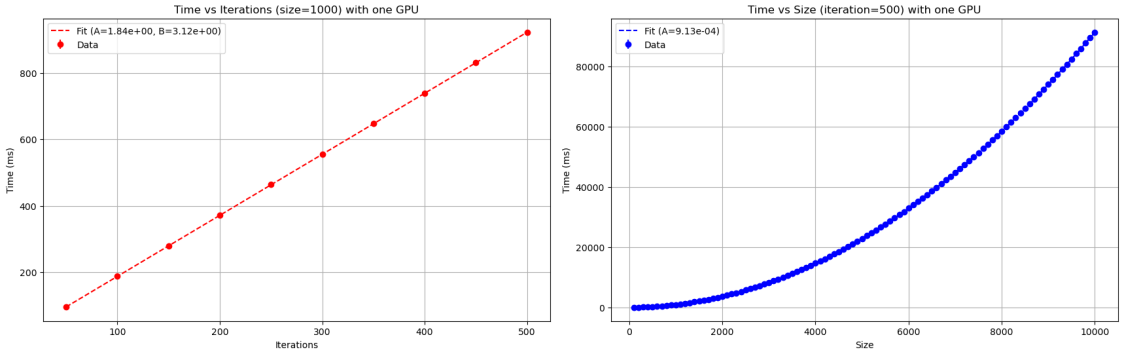
3.2.1 Launch file

Here, we show the main SLURM's parameters used to launch the program:

```
#SBATCH --partition=cuda
#SBATCH --time=00:10:00
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --mem-per-cpu=1G
#SBATCH --gres=gpu:1
```

3.3 Performance

We analyzed how execution time is influenced by variations in grid size and the number of iterations (i.e., the number of generations). To illustrate these findings, we created the following two graphs:



As observed, the execution time exhibits a linear trend when the number of iterations is varied. Conversely, the second graph reveals that execution time increases quadratically with the grid size. This behavior is consistent with the fact that the number of grid cells grows quadratically as the grid's side length increases.

3.4 Improvement: 2 GPUs

It is possible to improve the performance of the algorithm by splitting the Conway grid into two sub-grids of equal size. Then, computing the sub-grids with two separate GPUs, the overall computation time drastically decreases. The strategy implemented to achieve such is explained in the following lines.
The starting point is splitting the grid in two, we divided it into an upper and lower region, because it is easy to implement when working with a 1-dimensional pointer for the 2-dimensional matrix. Then the CPU process is split in two OpenMP threads, as:

```
#pragma omp parallel firstprivate(iter) shared(start, stop, elapsedTimePtr)
{
    int tid = omp_get_thread_num();
    ... Single GPU algorithm...
}
```

where the variables `start`, `stop` are used for the execution time recording, `iter` is the number of iteration to perform and `elapsedTimePtr` is the pointer to the variable that will contain the total execution time. Each separate OpenMP thread is handling a different part of the original Conway grid with their own GPU. Then the algorithm is the same as the one used with a single GPU, with the exception that the GPUs must be synchronized at every new iteration. Whenever the kernel returns, the two OpenMP threads must be put in sync, then the halo exchange, between the two GPUs, of the border cells must be performed, only then the new iteration can start.

```
#pragma omp barrier
if (tid == 0) {
    cudaMemcpyPeer(previous_matrix[1],
                   1,
                   previous_matrix[0] + n*(n/2 - 1),
                   0,
                   n * sizeof(int));
    cudaMemcpyPeer(previous_matrix[0] + n*(n/2),
                   0,
                   previous_matrix[1] + n,
                   1,
                   n * sizeof(int));
}
#pragma omp barrier
```

It is important to note, how in the code the halo exchange is performed only by a single thread, moreover, both threads must be aligned before entering and after exiting the latter section of the code.
It is important to change a parameter in the SLURM file to obtain access to two GPUs:

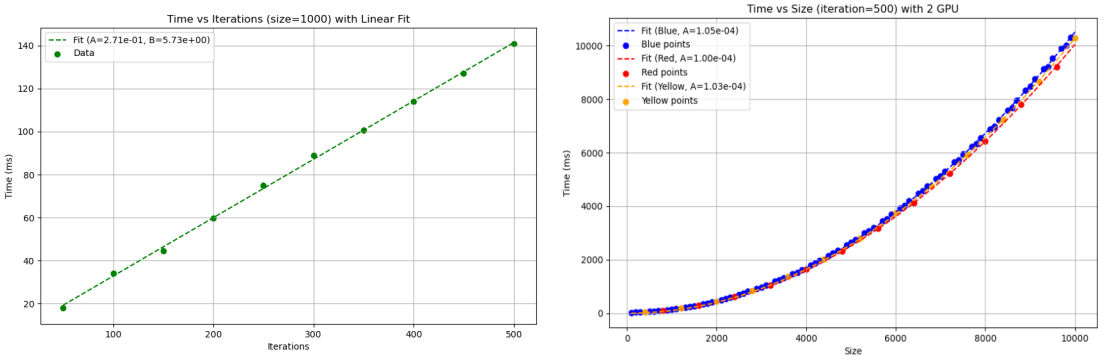
```
#SBATCH --gres=gpu:2
```

and when compiling with `nvcc`, but also OpenMP is used, one must compile in the following way:

```
nvcc -O3 -Xcompiler="-fopenmp" main.cu -o main_cuda
```

3.4.1 Performance

Similarly to the single GPU case, we examined how execution time is affected by changes in grid size and the number of iterations. To present these results, we generated the following two graphs:



As in the previous case, the execution time displays a linear pattern as the number of iterations changes. In the second graph, the data points are grouped into three distinct sets, each following a different parabolic curve. Notably, the red points exhibit slightly shorter execution times compared to the blue and orange points. Probably this difference is due to an alignment between the grid size and the internal architecture of the two GPUs.

4 Single node: OpenMP

4.1 Why OpenMP

OpenMP is a very efficient solution for parallelization problems when it is possible to use only one node unequipped of GPU. This framework can be used inside a C file and it makes parallelization really easy to perform and understand. Its advantages are also in the fact that the possible issues like resource contention, deadlock etc. are automatically managed during compilation. It is possible to extend the parallelization depending on the number of cores of the CPU.

4.2 Parallelization strategy

In OpenMP it is possible to perform the parallelization of a for loop, it is very useful when the code inside the loop must perform operations that do not depend on the results previous ones. In this case we achieve this parallelization by using the static scheduling writing the following line before the for loop

```
#pragma omp parallel for schedule(static)
for (int j = 0; j < n * n; j++){
    ...do something...
}
```

OpenMP will now assign to each of the cores a proper number of tasks, each one consisting in the execution of an iteration block with a specific value of index j. This type of parallel for is performed twice inside the code for each iteration, in fact, to compute each new generation, firstly the previous grid's data structure is updated with the new one using a parallel for and then another parallel for will perform the actual iteration generating the new grid. This type of implementation is done for synchronization purposes and in the attempt of speeding up the updating process of the previous matrix.
The code will look like this:

```
for (int i = 0; i < num_iter; i++) {
    #pragma omp parallel for schedule(static)
    for (int j = 0; j < n * n; j++)
        prev[j] = mat[j];

    #pragma omp parallel for schedule(static)
    for (int j = 0; j < n * n; j++) {
        ...game of life...
    }
}
```

4.2.1 Launch file

The SBATCH file to submit the job to SLURM in this case has two important parameters that must be properly set

```
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=12
```

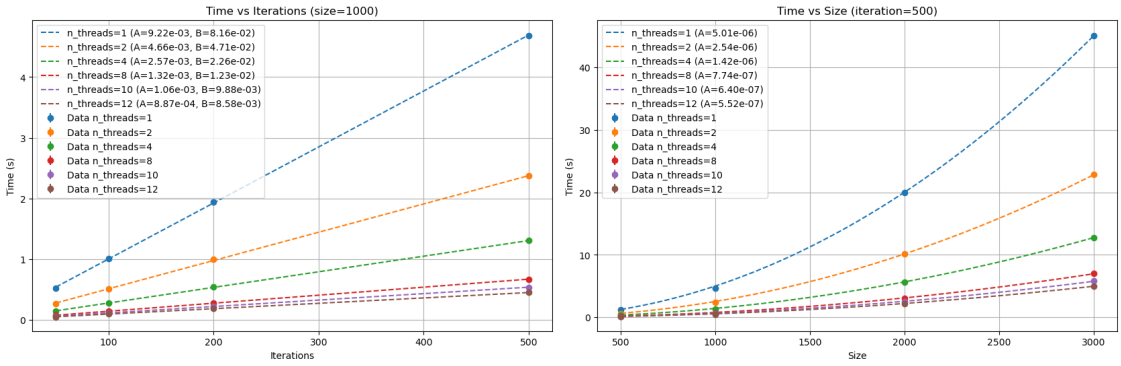
the number of tasks indicates just how many nodes are needed to work on the job, while the numbers of CPUs per tasks indicates how many maximum threads are used for the parallel execution, if just 8 threads are available in the node it is not possible to study the performance of parallelization with an higher number of cores.

Moreover the SBATCH file is written in such a way that statistics can be gathered using different number of threads on different initial grids randomically generated using python, overall 10 different runs are made for each thread, iterations and size configurations. The command for the file compilation uses an optional parameter "-O3" which performs several static optimizations improving the performances.

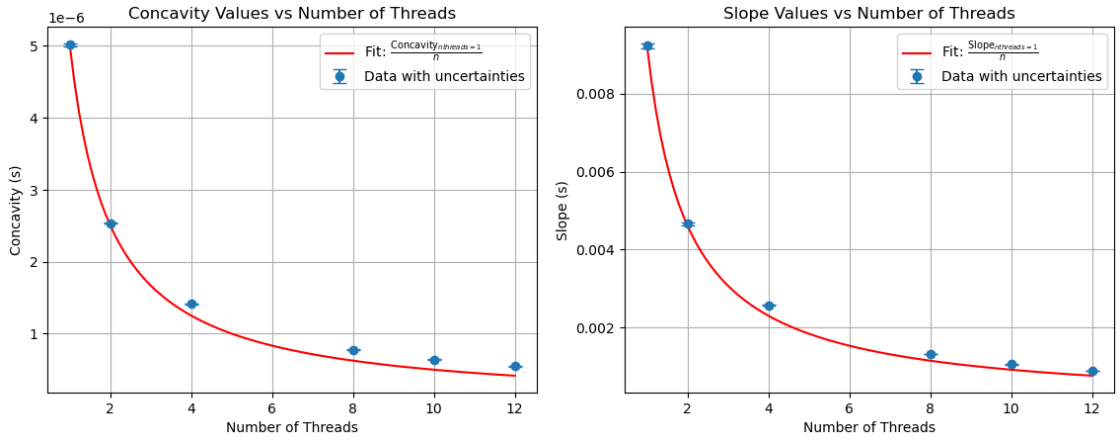
```
gcc -fopenmp -O3 -o main_omp main_omp.c
```

4.3 Performance

As in the previous sections, we explored how execution time is affected by changes in grid size and the number of iterations. To present these results, we generated the following two graphs:



In this case as well, the execution time follows a linear trend when the number of iterations is varied and a quadratic trend when the grid size changes.
To examine the impact of the number of threads on execution time, we analyzed the slope and concavity of the lines and parabolas shown in the previous graphs. Specifically, we plotted these values with the number of threads on the x-axis, yielding the following results:



We observe that the data follows a hyperbolic trend, consistent with the expected scaling of execution time as $\frac{1}{\texttt{n\_threads}}$. However, the observed decrease in execution time is slightly slower than predicted by the fit. This discrepancy is likely due to time overhead caused by communication between threads.

5 Multiple nodes: MPI

5.1 Why MPI

MPI allows distributed computing in architectures with multiple separated nodes by proividing an interface for the communication between the nodes. This solution allows to efficiently parallelize algorithms when we have more then one node available. In this case since Hactar is provided with multiple nodes it is possible to test the performances of this solution.

5.2 Parallelization strategy

The idea is to subdivide the grid in different subgrids composed by a subset of their rows, in this way a single subgrid is given to a node which will perform the game iteration in that part of the grid. If a grid has N rows and we are using m MPI nodes at node with rank 0 will be assigned the first N/m rows, while at rank 1 the next N/m rows and so on, in this way the workload is equally distributed among the different nodes. But, in order to correctly compute the new first and last rows of a subgrid it is necessary to have previous and next rows of a subgroup, to solve this issue it is necessary a mutual exchange of rows between the different nodes before each iteration. The initial subgrids are distributed by rank 0 which is also storing the complete data structures, during the iterations the exchange of rows is independently managed by the single nodes, while after the last iterations all the subgrids are gathered by the rank 0.

```
if (rank==0)
   for (int r=1; r<numtasks; r++)
        ...
        MPI_Send(&mat[offset], rows * n, MPI_INT, r, 0, MPI_COMM_WORLD);
else
    MPI_Recv(local_mat, rows_per_process * n, MPI_INT,
            0, 0, MPI_COMM_WORLD, &Stat);

for(int t=0; t<num_iter; t++){
    ...update local grid...
    MPI_Barrier();
    ...MPI_Isend() and MPI_Irecv() for rows exchange...
    MPI_Waitall();
    game_of_life();
}
MPI_Barrier();
MPI_Gatherv(&local_mat[0], rows_per_process * n, MPI_INT, &mat[0], recvcounts, displ
```

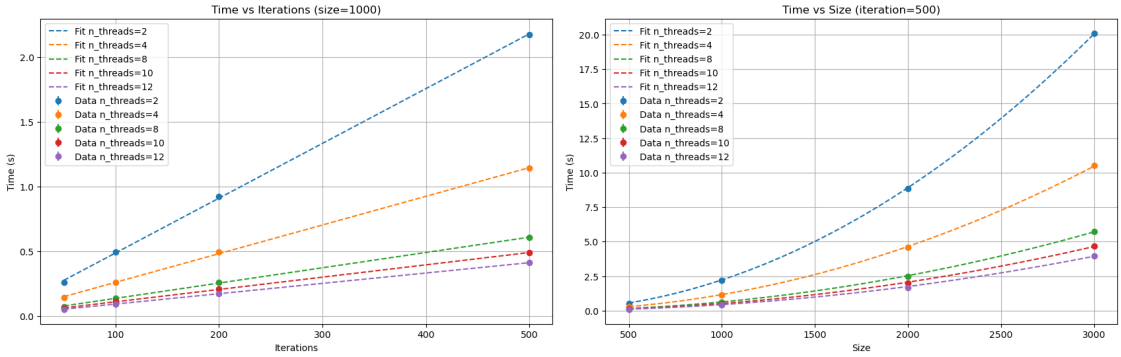
5.2.1 Launch file

The SBATCH parameters are opposite with respect to the OpenMP case, since here we are parallelizing on number of nodes and not on CPUs. The compiler mpicc has the parameter "-O3" with the same working principles of gcc.

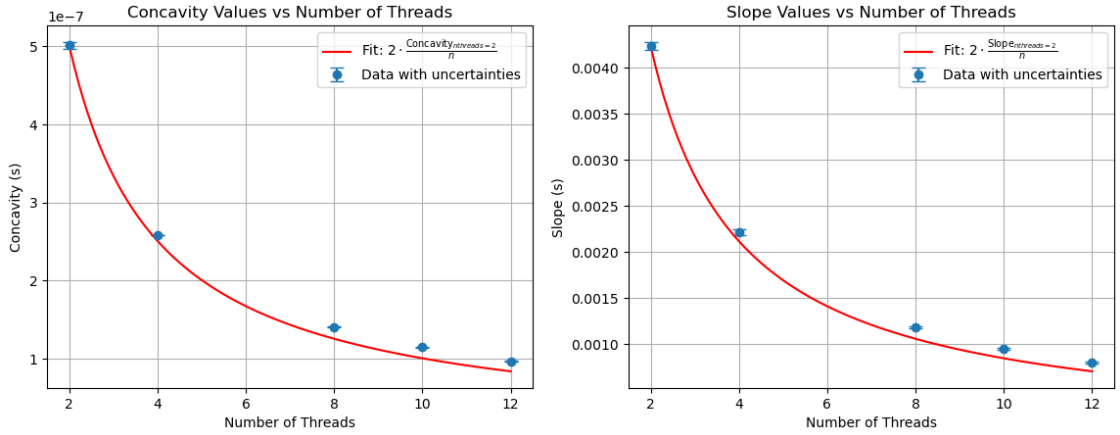
```
#SBATCH --ntasks=6
#SBATCH --cpus-per-task=12
mpicc -fopenmp -O3 -o main_hyb main_hyb.c
```

5.3 Performance

We again analyzed how execution time is influenced by variations in grid size and the number of iterations. We can see the outcome in these graphs:



Similarly, in this case, the execution time shows a linear pattern when the number of iterations is varied and a quadratic pattern when the grid size is modified.
Also here we analyzed the effect of the number of threads on execution time by examining the slope and concavity of the lines and parabolas. The resulting graphs are shown below:



Even in this case, our data exhibits a hyperbolic trend, with a slight discrepancy as the number of threads increases.

5.4 Integration with OpenMP

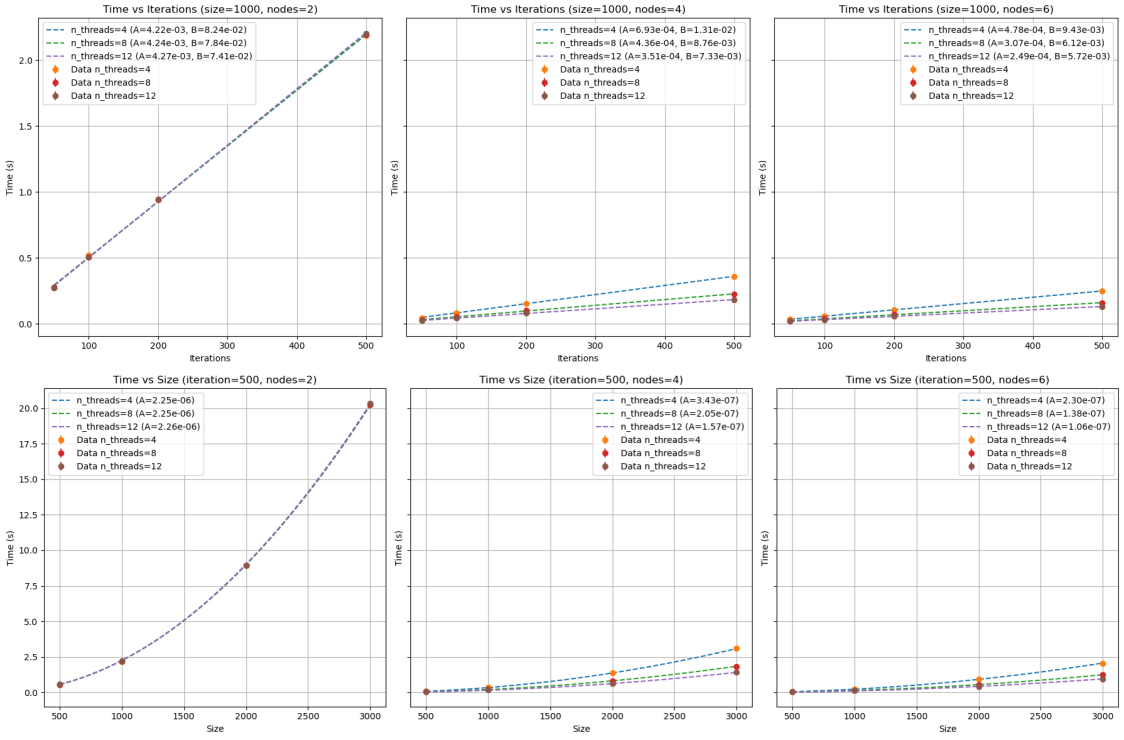
5.4.1 Idea

MPI distributes the computations among the different nodes and speeds up the execution but the workload for each node is executed completely in a sequential manner. The idea is to parallelize the execution on single nodes using OpenMP, this should guarantee a further improvement on the performances.
The implementation requires a very small modification on the original MPI code. In the *game_of_life()* function the principal loop is parallelized using a static schedule.
In this C code is possible to pass as command line arguments the number of MPI nodes and the number of OpenMP threads per node. Of course the SBATCH must be modified accordingly.

```
#SBATCH --ntasks=6
#SBATCH --cpus-per-task=12
mpicc -fopenmp -O3 -o main_hyb ../src/main_hyb.c
```

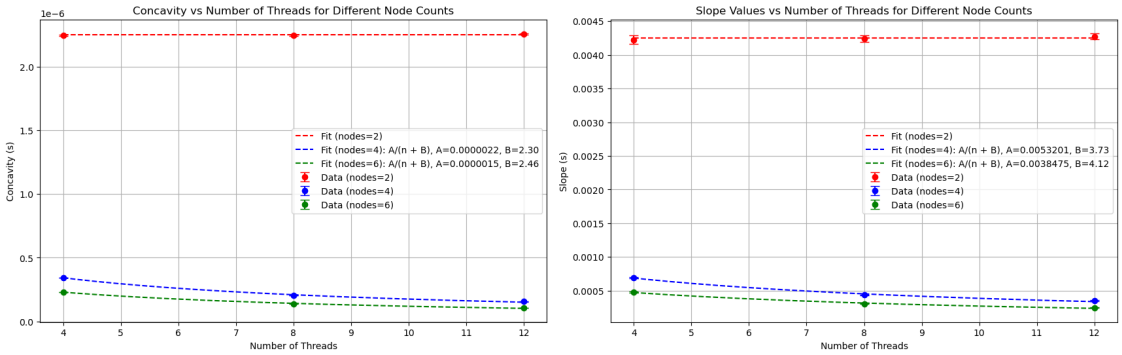
5.4.2 Performance

As in the previous sections, we analyzed how execution time is affected by changes in grid size and the number of iterations. However, in this case, we examined this trend across different numbers of nodes. To present these findings, we generated the following six graphs:



For all the nodes analyzed, the execution time follows the expected linear trend when varying the number of iterations and the quadratic trend when changing the grid size. There are

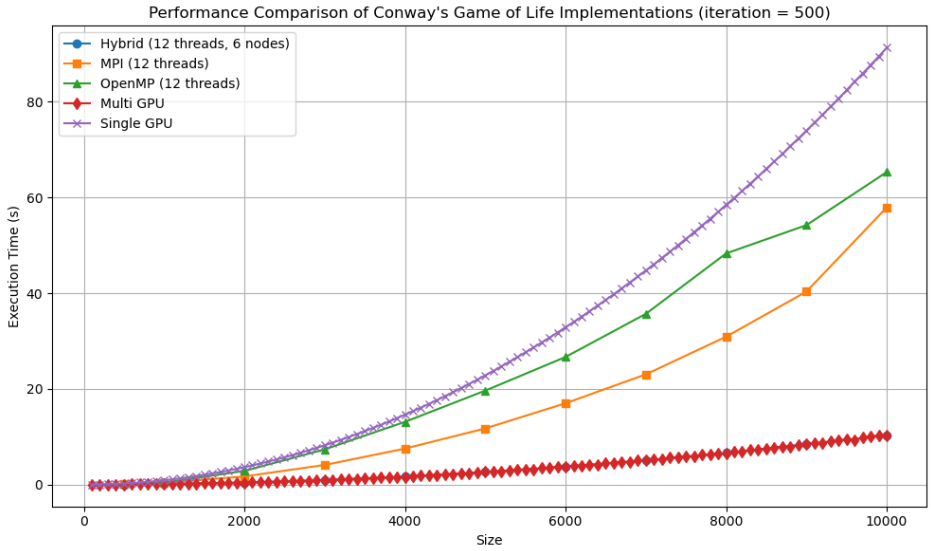
some noteworthy features that emerge when analyzing the trends of the slope and concavity as the number of threads varies. To highlight these patterns, we analyzed for any number of nodes the slope and concavity of the lines and parabolas shown in the previous graphs. Specifically, we plotted these values with the number of threads on the x-axis, yielding the following results:



With two nodes, there is no notable improvement in the program's performance. However, as the number of nodes increases, the execution time follows the expected hyperbolic trend.

Results

In conclusion, we analyzed the performance of various implementation techniques for Conway's Game of Life, focusing on execution time relative to grid size over 500 iterations. The resulting graph highlights the differences in efficiency and scalability across the tested approaches:



The Hybrid implementation (12 threads, 6 nodes) and the CUDA implementation with 2 GPUs exhibit similar performance, consistently achieving the best results across all input sizes. These two approaches significantly outperform the others, showcasing their efficiency and scalability for computationally intensive tasks. In contrast, the MPI and OpenMP implementations show moderate scalability, while the CUDA Single-GPU setup lags behind, particularly with larger inputs.