*Instituto Superior Técnico*

**Professors:**
José Monteiro
Afonso Alemão
Catarina Bento
Miguel Graça

**Course:**
Parallel and Distributed Computing

---

# Parallel and Distributed Computing - Game Of Life 3D MPI Delivery

**Group 27**

---

***Authors:***

Tomás Marques, 99340
Miguel Mano, 99286
João Ivo, 90115

March 30, 2024

# 1 Introduction

This report, made in the context of the Parallel and Distributed Computing course's MPI project assignment, serves to explain our approach to the parallelization and redesign of the previous serial and OMP versions. The project is based on the Game of Life with a 3D matrix composed by cells either dead or alive, carrying a value corresponding to one of the species in case of the cell being alive. Our task was to iterate over the 3D Matrix cube and simulate new generations. There were some changes made to the last serial version that resulted in better execution times and are explained in the OpenMP report on the omp folder. This report only includes details and results about the hybrid MPI + OMP solution.

# 2 Hybrid Approach

The final version of the program uses a hybrid solution that combines MPI and OMP. While the OMP approach hasn't changed, with only the simulation loop being parallelized for the reasons presented in the previous report, the MPI approach required substantial changes to our base code. These are the steps that the program follows upon starting, to allow for parallel execution:

- Creation of a 2D Cartesian topology and assignment of coordinates to processes according to rank;
- Calculation of up, down, left and right neighbors for each process on the Cartesian grid (with wrap-around);
- Calculation of partition of grid belonging to each process, according to Cartesian coordinates;
- Allocations and initialization of specific part of grid for each process, according to partition data;

This approach ensures that each process only allocates memory and initializes the part of the grid that it will work with. A detail to take into consideration is that each processes' grid has the same length in the Z dimension as the other processes, meaning the original grid is only split in the X and Y dimensions since the Cartesian topology is 2D. Furthermore, since each grid may not be evenly divisible among the processes, the remainder of X coordinates are attributed to processes in the first row of the Cartesian topology and the remainder of Y coordinates are attributed to processes in the first column of the topology. Figure 1 shows an example of this process with four processes and a grid of side five. Finally, when allocating memory for the grid, two extra units are given to the X and Y dimensions, in order to have space to store the "ghost edges" that are shared among processes and used to process cells at the borders of partitions. This process is explained in more detail later in the report.

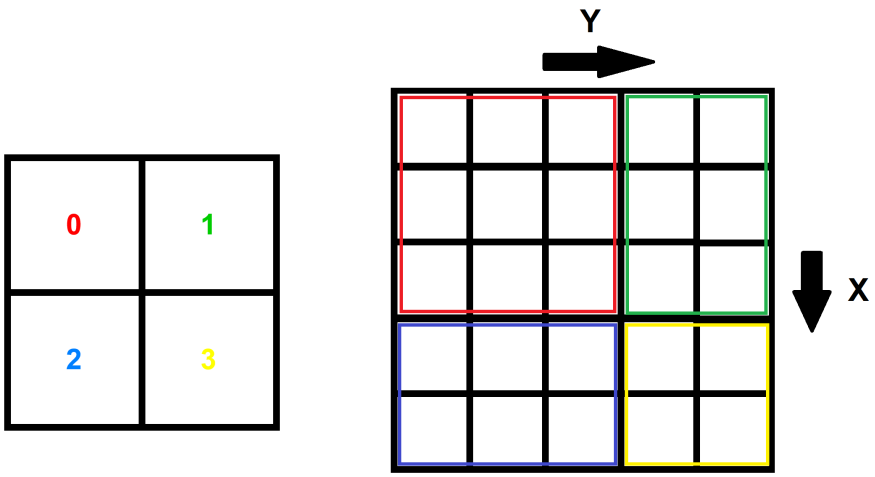


Figure 1: Mapping of processes to grid partitions in the X and Y dimensions.

The simulation process remains largely unchanged in the actual calculation of dead/alive cells for the next generation, with the only difference being present in the computation of the neighbors of a cell. Since each process has access to "ghost edges" around its grid that contain the border cells of neighboring processes, there is no need to perform wrap-around calculations on the X and Y axis and these are performed only for the Z axis.

While the calculation methods have remained the same, it is now necessary to perform a few steps before getting results, since each process needs to have information about the grids of neighboring processes, specifically the cells on the borders. These bordering cells are shared between processes horizontally and vertically and stored in the extra allocated space of the grid of a process, which we have called "ghost edges", since the cells in these regions are only used for neighbor calculations and don't actually belong to the grid partition. Here is the process in more detail and order of execution:

**Horizontal edges**

- Each process calculates the Y and Z positions of the alive cells in its upper and lower borders;
- Upper border info is sent to neighbor up and lower border info is sent to neighbor down;
- Each process places the received alive cells from above in the correct Y and Z positions of its upper "ghost edge" and the received alive cells from below in the correct Y and Z positions of its lower "ghost edge";

**Vertical edges**

- Each process calculates the X and Z positions of the alive cells in its left-most and right-most borders, including the cells on the upper and lower "ghost edges" added on the previous step (but still on the left and right border);
- Left border info is sent to left neighbor and right border info is sent to right neighbor;
- Each process places the received alive cells from the left in the correct X and Z positions of its left-most "ghost edge" and the received alive cells from the right in the correct X and Z positions of its right-most "ghost edge";

These steps ensure that all processes get the necessary border cells information while also avoiding contacting diagonal processes, due to the inclusion of the "ghost edge" cells in the first step of the vertical edges. All send and receive operations are non-blocking in order to improve the performance of the program but this then requires that the processes should wait to receive all horizontal edges before adding them to its "ghost region" and the same for the vertical edges later on. When all " ghost edges" have been added to the grid, the simulation is performed on each process, the "ghost edges" are cleared (set to zero) for a future generation and the results of the generation of each process are summed and sent to the root process which is responsible for printing the final results and execution time at the end of the program run.

# 3 Results and Conclusions

## 3.1 Execution Times and SpeedUps

To measure the execution time of our project we used the routine *omp_get_wtime* (only measuring the execution time of the simulation through the generations), and tested the program in the PCs of Lab 2 RNL (6 cores, 2 threads per core). For our analysis testing sample, we've used the four input examples available at the course project main page, and measured their execution time with varying numbers of processes and threads, with the following command:

"srun -C lab2 -n <processes> –cpus-per-task=<threads p/ process> life3d-mpi gens n_cells density seed"

The tables contain our execution times in seconds, in order of smallest to highest average speedup when compared to the serial version. The samples are identified by the number of cells per side of the cube. For the speedup calculations, we've used the *Amdahl's Law* speedup calculation, using the serial time. We also present tables with the average speedups of each test.

| Samples | Serial | 1P 1T | SpeedUp | 2P 1T | SpeedUp | 1P 4T | SpeedUp | 4P 1T | SpeedUp | 2P 4T | SpeedUp |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 64 | 16.8 | 13.6 | 1,24 | 12.3 | 1,37 | 6,4 | 2,62 | 6,2 | 2,71 | 4 | 4,2 |
| 128 | 27.4 | 22 | 1,25 | 19,7 | 1,39 | 10 | 2,74 | 9,9 | 2,77 | 5,5 | 4,98 |
| 512 | 107.9 | 87.2 | 1,24 | 72,9 | 1,48 | 36,8 | 2,93 | 36,4 | 2,96 | 20,3 | 5,32 |
| 1024 | 319 | 250.7 | 1,27 | 209,2 | 1,52 | 106 | 3,01 | 104,7 | 3,05 | 80,5 | 3,96 |

| Samples | 8P 1T | SpeedUp | 4P 4T | SpeedUp | 16P 1T | SpeedUp | 32P 1T | SpeedUp | 8P 4T | SpeedUp |
|---|---|---|---|---|---|---|---|---|---|---|
| 64 | 3.7 | 4,54 | 2,8 | 6 | 2,3 | 7,3 | 1,5 | 11,2 | 2 | 8,4 |
| 128 | 5,3 | 5,17 | 3 | 9,13 | 3 | 9,13 | 1,7 | 16,12 | 1,7 | 16,12 |
| 512 | 20,2 | 5,34 | 12,1 | 8,92 | 11,9 | 9,07 | 8,8 | 12,26 | 6 | 17,98 |
| 1024 | 79,3 | 4,02 | 47,2 | 6,76 | 43,9 | 7,27 | 26,5 | 12,04 | 20,9 | 15,26 |

| Samples | 64P 1T | SpeedUp | 16P 4T | SpeedUp |
|---|---|---|---|---|
| 64 | 1,1 | 15,27 | 1,1 | 15,27 |
| 128 | 1 | 27,4 | 1 | 27,4 |
| 512 | 5 | 21,58 | 3,6 | 29,97 |
| 1024 | 17,6 | 18,13 | 12,2 | 26,15 |

| — | 1P 1T | 2P 1T | 1P 4T | 4P 1T | 2P 4T | 8P 1T |
|---|---|---|---|---|---|---|
| Avg SpeedUp | 1.25 | 1.44 | 2.83 | 2.88 | 4.62 | 4.77 |

| — | 4P 4T | 16P 1T | 32P 1T | 8P 4T | 64P 1T | 16P 4T |
|---|---|---|---|---|---|---|
| Avg SpeedUp | 7.7 | 8.19 | 12.9 | 14.44 | 20.6 | 24.7 |

When analysing the results, we can see that the speedups do not equal the amount of processes and threads. For example, for 1 process and 4 threads (or 4 processes and 1 thread), the speedup is around 2.8 and not close to 4 like in the purely OMP version of the project. This reduction in speedup may be happening due to the increased overhead introduced by the communication of the processes before performing the simulation. Nonetheless, we can observe that the speedup keeps increasing as the number of processes and threads gets higher. Interestingly, the speedup is similar when the number of "tasks" (processes x threads) is the same (e.g. [2P 4T] and [8P 1T]) except for 32 and 64 "tasks", where it seems to be better to have a lower amount of processes but a higher amount of threads per process, again, possibly due to the reduced amount of communication between processes. This indicates that a hybrid solution is better than a purely MPI one. Finally, it's also worth highlighting that the solution has speedups even when only 1 process and 1 thread is used. This may be happening due to the elimination of wrap-around neighbor calculations for the X and Y axis, which are now calculated through the communication of "ghost edges" as described in the previous section.