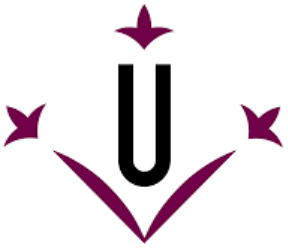

Universitat de Lleida

Implementation with MPI of the HPC project

Made by
Oriol Alàs Cercós, Sergi Simón Balcells

Delivery
1st of June, 2022

Universitat de Lleida

Escola Politècnica Superior

Màster en Enginyeria Informàtica

High Performance Computing

Professorate:

Jordi Ricard Onrubia Palacios

Contents

List of Figures

List of Tables

1 Introduction

The project consists of developing the game of life program in a parallelized environment using MPI. The first sections will reason about which parallel algorithms we could use, which one we choose, and why we did it. We will also discuss the project structure, as a single file was not readable when developing with MPI.

Then, we will provide the results of this code executed at the Moore cluster with the examples of 5000 and 10000 matrices provided in the assignment. In the following section, we will analyze why the application behaves like that.

Finally, a conclusion to the analysis is provided, where we remark on the key points that were discovered in it.

2 Algorithm Selected

The algorithm could take three shapes depending on which communication the nodes use:

1. Divide the square into smaller squares. This solution is inefficient in terms of communication, as it has to send and receive data from 4 other cores, as seen in the figure 1.

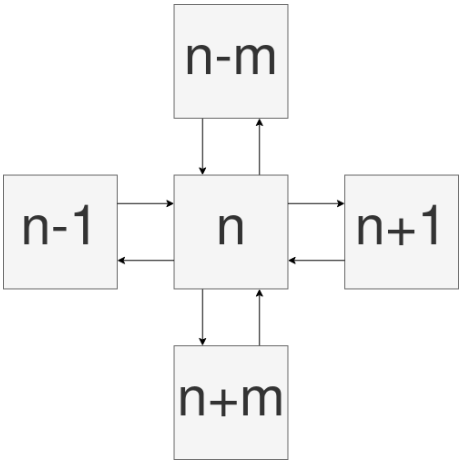

Figure 1: Matrix figure to showcase the algorithm. In this example, a node connects to all the four neighbors.

2. Divide the matrix into columns. This solution is better than before, as it only sends the information to two other nodes, making the transportation less painful.

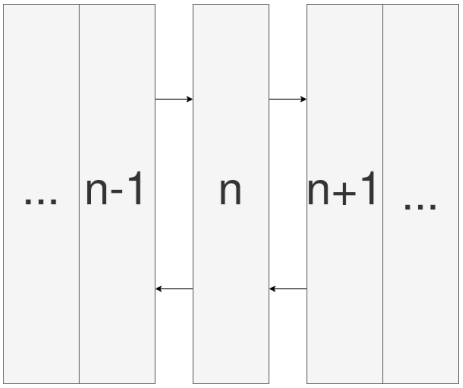

Figure 2: Columns figure to showcase the algorithm. In this example, a node connects to two neighbors

3. Divide the matrix into rows. The same as the last one, but it has better memory allocation, as the items are next to each other in the memory.

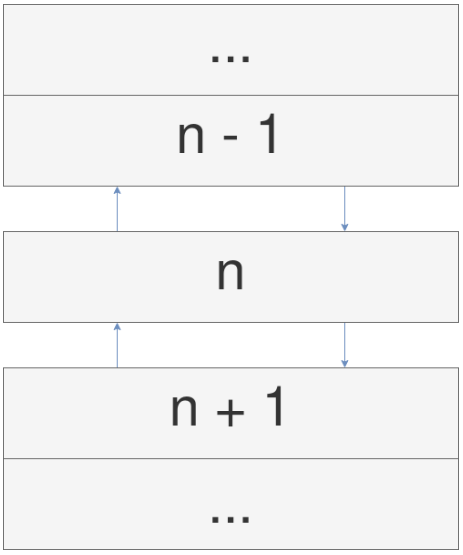

Figure 3: Rows figure to showcase the algorithm. In this example, a node connects to two neighbors

So, for said reasons, we will be developing our program with the communication orientated to rows.

Although MPI is a relatively high-level interface, it does not provide as much abstraction as one would desire, so there is more boilerplate code to achieve the parallelization of the program. As such, the code quickly became a hassle to navigate, so it was divided into some other modules, which are:

`contants.h` This module contains all the constants that the program uses, as the debug

constant. All the constants are declared as `const type` if they are not used by arrays, to avoid the VLAs.

`debug_tools.h` Inside this file some utilities only execute if the debug option is set to true. This, combined with an optimization 3 on the compiler makes all the debug functions disappear in the end code.

`mpi_utilities.h` Most of the commands to send and receive from the MPI are repeated through the code. This file contains an abstraction layer to avoid them.

`utilities.h` Some functions that the professor provided us.

`read_life.h` This file contains all the functions regarding the reading or creation of the matrix to perform the game of life, as well as the necessary communication with all the nodes for their table.

`update_file.h` This module contains all the functions related to doing an update of the game of life. It also contains all the necessary communication between these modules.

`main.c` Contains the executable, as well as some little functions.

Finally, the compilation of this program was:

```
mpicc -o main main.c -O3 -std=gnu99
```

Where O3 activates the optimizations, it also is specified to compile the source with gnu99, as it used a non-compatible version with the code by default.

3 Results

The results are gathered in the table 1, 2, 3. There are different graphics that can be seen in these application, resuming the fact of how evolves the time, the speed-up, and the efficiency in the figures 4, 5, 6, 7 respectively.

The time measured in these graphics and tables is the time that the application needs to read the file and update until the maximum number of iterations is met.

Table 1: Time table. There are two examples used in this process, both being square matrixes of 5000 and 10000 specifically.

cores	5000	10000
1	325.84	2550.46
2	243.86	1428.25
4	117.93	1046.93
8	59.82	514.27
16	30.95	239.19
32	16.81	123.97

Table 2: Speedup table. There are two examples used in this process, both being square matrixes of 5000 and 10000 specifically.

cores	10000	5000
1	1.00	1.00
2	1.79	1.34
4	2.44	2.76
8	4.96	5.45
16	10.66	10.53
32	20.57	19.39

Table 3: Efficiency table. There are two examples used in this process, both being square matrixes of 5000 and 10000 specifically.

cores	10000	5000
1	1.00	1.00
2	0.89	0.67
4	0.61	0.69

Continued on next page

cores	10000	5000
8	0.62	0.68
16	0.67	0.66
32	0.64	0.61

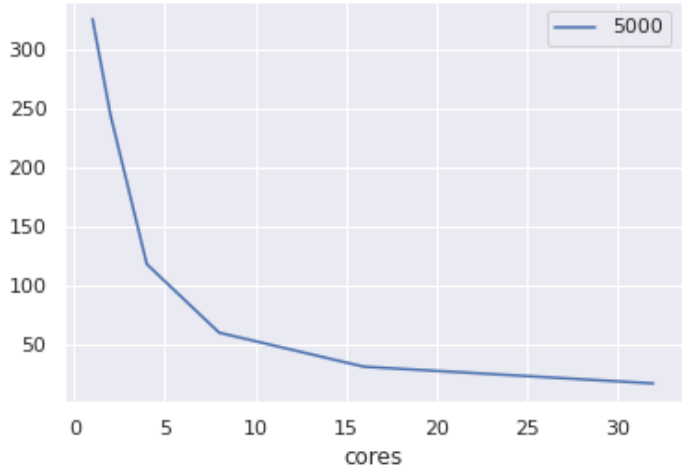

Figure 4: Time plot. The y axis is the time in seconds while the x axis is the number of cores used. This is only for the 5000 sized matrix.

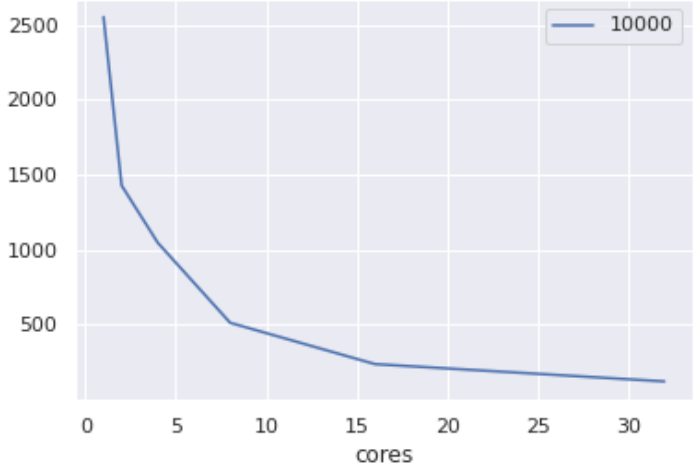

Figure 5: Time plot. The y axis is the time in seconds while the x axis is the number of cores used. It contains the data from the 10000 sized matrix.

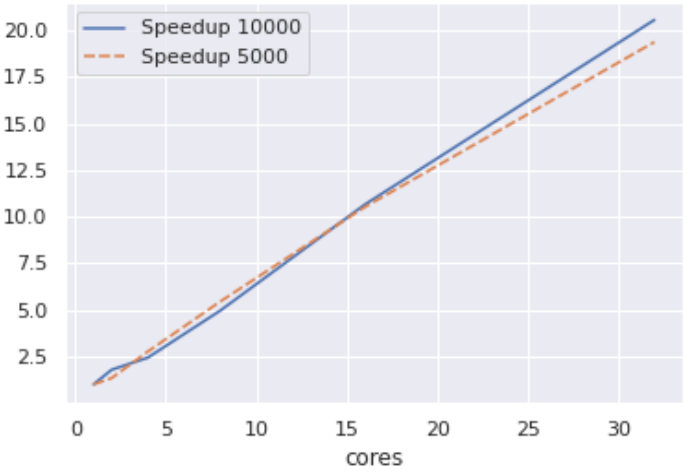

Figure 6: Speedup plot. The y axis is the speedup while the x axis is the number of cores used.

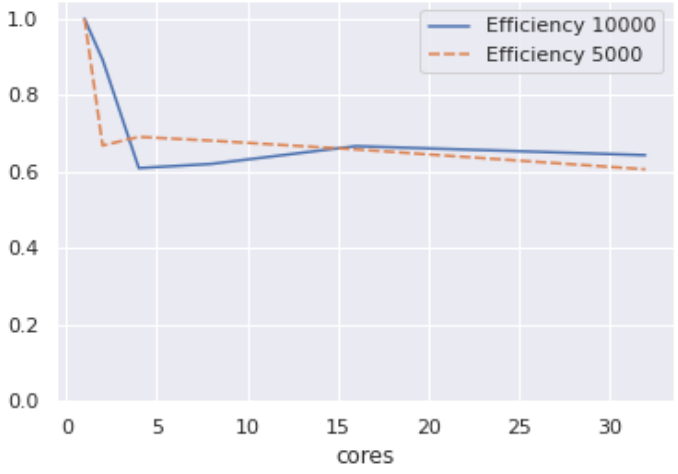

Figure 7: Efficiency plot. The y axis is the efficiency while the x axis is the number of cores used.

3.1 Analysis

This application is almost embarrassingly parallel, as the costs and the speedup increase linearly with the number of cores that uses them. But, there is a small bottleneck that is reading the file: as the disc storage is not concurrent, the reading must be set to read in parallel. Furthermore, For this reason, it can be seen that the efficiency worsens in each core added, as that part of the program is sequential.

This pattern can be seen more clearly at the efficiency plot, as the efficiency limits to the part of the code that can not be done concurrently, as shown in the next formula:

$$\lim_{x\to\infty} E(x) = \lim_{x\to\infty} \frac{T_s}{(T_{\text{read file}} + \frac{T_p + T_{\text{communication}}}{x}) \cdot x}$$

$$= lim_{x\to\infty} \frac{T_s}{T_{\text{read file}} \cdot x + T_p + T_{\text{communication}}} = 0$$

This means that the efficiency will eventually reach 0, as the time spend reading the file, and communicating with the other two nodes will never be parallelized. This also means that the maximum speed up is:

$$\lim_{x\to\infty} S(x) = \lim_{x\to\infty} \frac{T_s}{T_{\text{read file}} + \frac{T_p + T_{\text{communication}}}{x}} = \frac{T_s}{T_{\text{read file}}}$$

What's more, this pattern is shockingly the same with both benchmarks that were tried, where the maximum difference in speedup is 1.18, and both speedups are close to each other. With the median of 100 executions, this difference may be even lower. Furthermore, the efficiency also shows that the change when the overheat of communication as the number of nodes increases is maintained through all the executions, and it slowly decays as the reading of the file cannot be parallelized.

4 Conclusion

In this project we parallelized the algorithm of the Game of Life, reaching almost embarrassingly parallel levels of concurrency in the program with the correct use of the scheduler and the optimizations in the MPICC set to maximum

Moreover, it was determined that both the reading part of the game of life and the communication part is the only sequential part of the whole operation, which makes the efficiency slow down as the number of cores rises, where eventually it will reach 0. Additionally, the speedup is theoretically relative to the sum of the time spent reading the file, as well as the time spent communicating with two nodes.