Universitat de Lleida

Economic Viability

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 + OpenMP. This work is heavily based on the MPI program, so feel free to skip the 2nd section if you have read that work. 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.

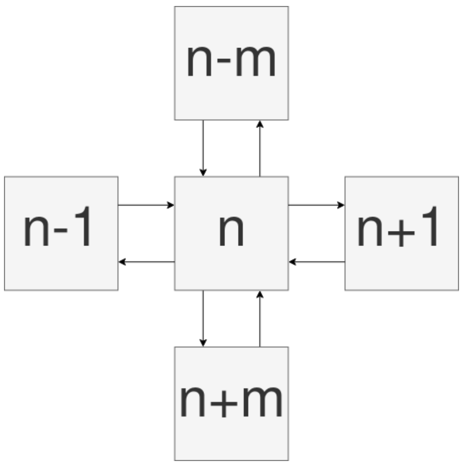

Figure 1: Matrix figure to showcase the algorithm. In this example, a node connects to all the four neighbors.

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.

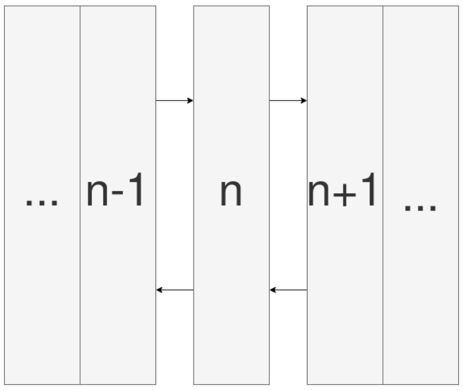

Figure 2: Columns figure to showcase the algorithm. In this example, a node connects to two neighbors

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.

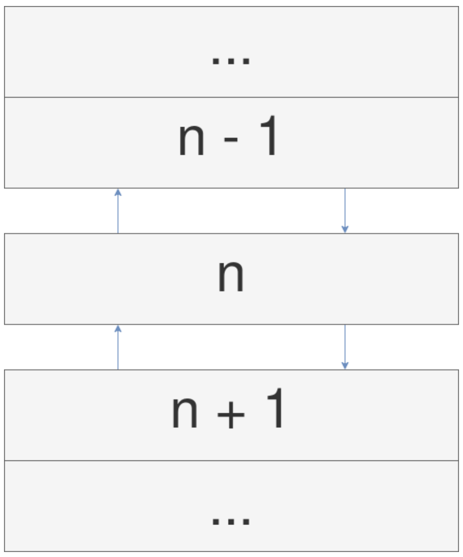

Figure 3: Rows figure to showcase the algorithm. In this example, a node connects to two neighbors

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 -O4 -std=gnu99
```

Where O4 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. Although 3 should be more than enough, but in some systems it does not work as intended.

3 Changes from the last algorithm

As usual, it was discussed the previous solution with the companions that delivered the work. One of the optimizations that we talked about, was the change on the types. As at most the support matrix can have 5, and the grid matrix can only either ones or zeros, using `int8_t` instead of 64 bits integers are faster, as the cost of the network is lower. There was a change that the optimized already did, which was to only allocate the support matrix once, and free it once the program finished. As we will comment on the analysis, the performance of the program was slower than it was expected, so we tried to optimize the code more. After a time searching, we discovered the reason of the slowness on the program, as it will be discussed on the results.

4 Results

The results are gathered separated by the size used. In each of them, the results are gathered between the threads that the OpenMP will use. The number of nodes used by the MPI is the number of cores (i.e. slots), divided by the number of threads of the application.

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 specifically.

	cores	1	2	4
0	1.00	477.68	nan	nan
1	2.00	245.50	250.33	nan
2	4.00	122.74	126.28	199.13
3	8.00	61.33	63.21	104.13
4	16.00	nan	32.20	59.65
5	32.00	nan	nan	36.33

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

	1	2	4
0	1.00	nan	nan
1	1.95	1.91	nan
2	3.89	3.78	2.40
3	7.79	7.56	4.59
4	nan	14.83	8.01

Continued on next page

	1	2	4
5	nan	nan	13.15

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

	1	2	4
0	1.00	nan	nan
1	0.97	0.95	nan
2	0.97	0.95	0.60
3	0.97	0.94	0.57
4	nan	0.93	0.50
5	nan	nan	0.41

Table 4: Time table. There are two examples used in this process, both being square matrixes of 10000 specifically.

	cores	1	4	2
0	1.00	3899.98	nan	nan
1	2.00	1959.40	nan	2004.72
2	4.00	964.54	1078.33	1012.20
3	8.00	482.75	734.53	501.03
4	16.00	nan	269.30	252.36
5	32.00	nan	203.55	nan

Table 5: Speedup table. There are two examples used in this process, both being square matrixes of 10000 specifically.

	1	4	2
0	1.00	nan	nan
1	1.99	nan	1.95

Continued on next page

	1	4	2
2	4.04	3.62	3.85
3	8.08	5.31	7.78
4	nan	14.48	15.45
5	nan	19.16	nan

Table 6: Efficiency table. There are two examples used in this process, both being square matrixes of 10000 specifically.

	1	4	2
0	1.00	nan	nan
1	1.00	nan	0.97
2	1.01	0.90	0.96
3	1.01	0.66	0.97
4	nan	0.91	0.97
5	nan	0.60	nan

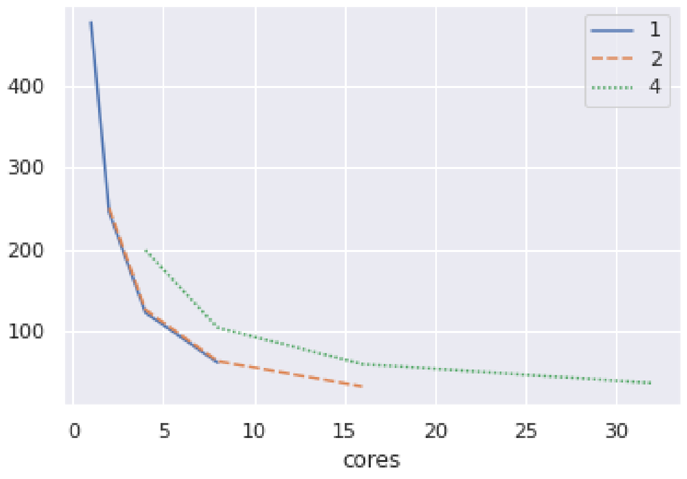

Figure 4: Time plot. The y axis is the time in seconds while the x axis is the number of cores used. This is only for the 5000 sized matrix.

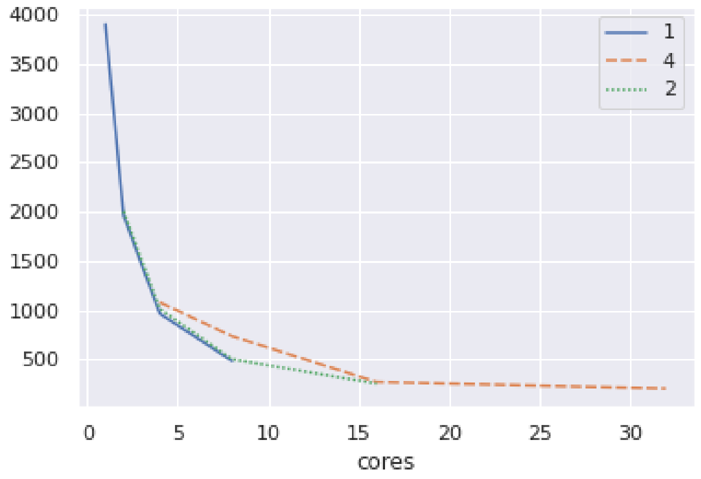

Figure 5: Time plot. The y axis is the time in seconds while the x axis is the number of cores used. This is only for the 10000 sized matrix.

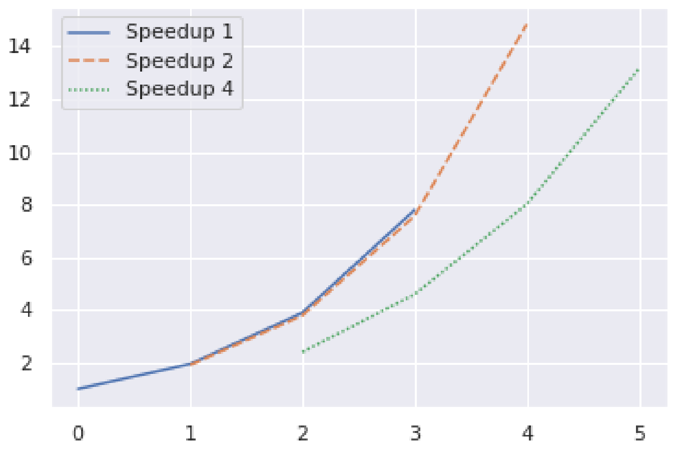

Figure 6: Speedup plot. The y axis is the speedup while the x axis is the number of cores used. This is only for 5000 sized matrix

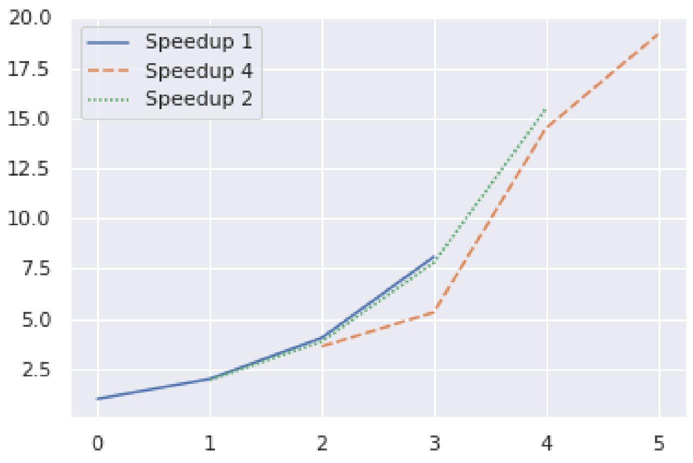

Figure 7: Speedup plot. The y axis is the speedup while the x axis is the number of cores used. This is only for 10000 sized matrix

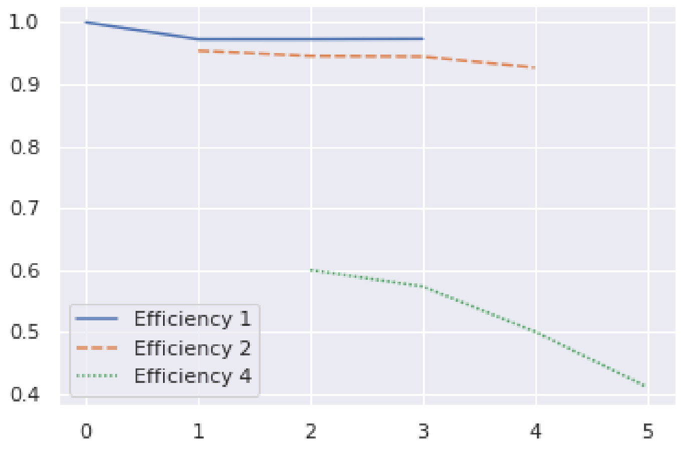

Figure 8: Efficiency plot. The y axis is the efficiency while the x axis is the number of cores used. This is only for 5000 sized matrix

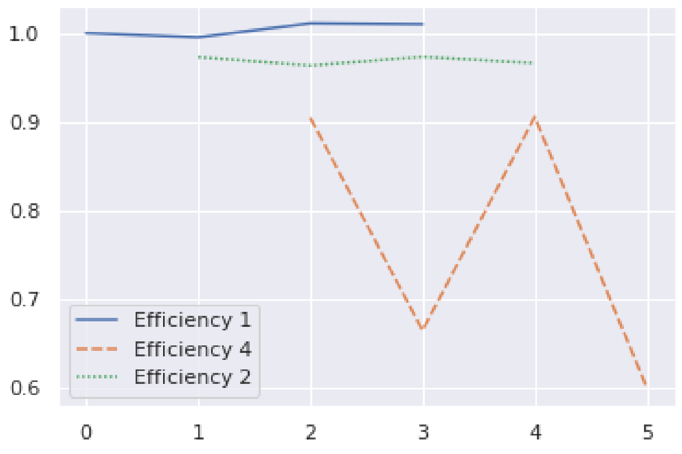

Figure 9: Efficiency plot. The y axis is the efficiency while the x axis is the number of cores used. This is only for 10000 sized matrix

4.1 Analysis

The costs analysis is the same as the previous work. As such we will divide the analysis between two sections, the formal analysis and a secondary analysis where we will analyze why OpenMP performs worse than the MPI counterpart. Feel free to skip the first section if you have already read the MPI project.

5 Formal 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.

6 Why MPI is faster

Hybrid computing is always faster than only using a MPI counterpart, as the communication between sockets is slower than the memory counterparts. Nevertheless, both MPI and OpenMP experts discourage using MPI + OpenMP, as [1], and [2] says. In a more understandable and concise way, when using optimizations with the compiler, the overheat of using both technologies shadows the more efficient approach of using a hybrid system. This is way MPI has included a threaded option, as well as some directives to make the hybrid approach faster, even if it is not as easy as OpenMP.

This is concluded when comparing the 32 slots using an hybrid approach versus using only MPI, where in this project it takes 32 seconds with the hybrid, and only 16 seconds with MPI, even without the memory optimizations.

7 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.

Nevertheless, it was stated that only MPI, with the optimizations of the compiler enabled is faster than the hybrid counterpart, as the overheat of using OpenMP overshadows the communication cost of MPI. It was stated that most probably using only MPI + MPI as an hybrid solution will be faster than all the solutions.

References

[1] *Rolf Rabenseifner, Georg Hager, Gabriele Jost.* **Hybrid MPI and OpenMP Parallel Programming**. `https://openmp.org/wp-content/uploads/HybridPP_Slides.pdf`, sl. 8.

[2] *Torsten Hoefler, James Dinan, Darius Buntinas, Pavan Balaji, Brian Barrett, Ron Brightwell, William Gropp, Vivek Kale, Rajeev Thakur.* **MPI + MPI: a new hybrid approach to parallel programming with MPI plus shared memory**. `https://rd.springer.com/article/10.1007/s00607-013-0324-2`.