1. Introduction

In the current study, the challenges of developing a parallel implementation of Conway's Game of Life, using the Message Passing Interface (MPI) in C, are presented, as well as the results in terms of measured times, speedups and efficiencies, compared to the sequential program. In section 2, the main design choices are discussed, regarding grid decomposition, splitting the world into subworlds for each process, communicating the boundary conditions – ghost cells – between neighbouring processes, and finally gathering the data for printing purposes. In section 3, an analysis of the performance metrics is conducted, for different numbers of DAS5 nodes and cores per node, both for a small and a large problem size. In section 4, a column wise grid decomposition is compared to the basic row wise one, and the differences between blocking and non-blocking communication in this specific application are discussed. Finally, in the last section, we reach to a number of conclusions regarding our implementation.

2. Implementation

The first challenge when it comes to the parallelization of the current application is to decide an efficient way to split the 2D world in smaller 2D subworlds and assign each subworld to a process. The goal is to split the board as equally as possible, so that each process has approximately the same workload. In the basic implementation, the world is split row wise, with each process receiving a number of complete rows. This is the role of the *grid_decomposition* function, which divides the total number of rows to the number of processes and if there are remaining rows, one more row is given to some processes, starting from process 0 and moving on until there are no remaining rows.

The next change, compared to the sequential algorithm, is that each process only allocates enough memory to fit its subworld, and not the whole world. Only process 0 allocates memory for the whole world, since it is the process that initializes the board, sends the parts at the corresponding processes and, at the end of each iteration, gathers the updated parts in order to print the whole board and the total number of alive cells. Regarding the splitting and gathering, collective MPI operations are used, namely *MPI_Scatterv* and *MPI_Gatherv*. The general (*v*) versions are used due to the possibility of unequal number of rows between processes. The *heights* array is used to store the number of rows of each process, and the *starts* array to store the starting row of each process. This information is required by the abovementioned MPI operations to scatter and gather the parts correctly. These operations are implemented in the *split_board* and *gather_updated_parts* functions accordingly.

Moving to the boundary conditions, inside the *world_border_wrap* function, the left-right ones are the same as in the sequential version. As for the top-bottom ones, each process gets the ranks of its neighbours with the *decide_neighbours* function and then sends its first real (not ghost) row to the top neighbour and the last one to the bottom neighbour, receives the last real row of its top neighbour and sets it as the top ghost row and the first real row of its bottom neighbour and sets it as the bottom ghost row. The top-bottom border wrap in the parallel program is taken into account with the top neighbour of process 0 being the last process and the bottom neighbour of the last process being process 0. In case of a single process, its neighbours are set as the process itself. As for the communication, both blocking and non-blocking primitives were tested. In the blocking MPI_Send/MPI_Recv approach, the even rank processes are receiving first, and the odd rank ones are sending first, to avoid a deadlock situation. As for the non-blocking approach, MPI_Isend/MPI_Irecv primitives are used and the computations of the inner part of the subworld, that does not require the ghost rows, are conducted, while the MPI_Requests are running, in order to overlap computation with communication. Then MPI_Waitall is used to wait for the MPI_Irecv requests to complete, before computing the top and bottom rows' new states. Finally, another MPI_Waitall is used for the MPI_Isend requests, before moving to the next iteration. The blocking approach seemed to be the most efficient in setups with a small grid and less than 8 nodes/cores, as well as in all the large grid cases. The non-blocking approach presented the best performance in setups with a small grid and 8 or more processes. However, the blocking approach was preferred, since it was the most efficient in most of the set ups, and especially in all the large grid ones. A more detailed analysis of blocking and non-blocking message passing is presented in section 4.

3. Performance Analysis

The results of all the simulations are presented below, in terms of measured times, speedups compared to the sequential algorithm, and efficiency. Regarding the measurements, only the time required to communicate the boundary conditions and compute the new states for all iterations was considered, leaving out world initialization, splitting, gathering and printing. Additionally, the elapsed time of the slowest process was always measured, by using an *MPI_Reduce* operation to get the maximum of the elapsed times of all processes. Speedups and efficiencies were computed using the following formulas:

$$speedup = \frac{measured\ time\ of\ the\ sequential\ program}{measured\ time\ of\ the\ parallel\ program\ with\ p\ processes}$$

$$efficiency = \frac{speedup}{p}\ 100\%$$

where p is the number of processes (nodes x cores).

The first experimental setup includes a small 1000x10 (rows x columns) grid and 20 million timesteps. The second one uses a large 1000000x1000 grid and 100 timesteps. The grid shape for the simulation of this implementation is chosen to be a rectangle with larger height than width (more rows than columns), since, after some testing, the row wise decomposition performed better in such grids. This fact was expected, since, this way, each process has more data to compute and less to communicate compared to a square grid, resulting to a higher computation/communication ratio. A different implementation with a column wise decomposition and the comparison between the two types of world splitting is discussed in section 4.

All the combinations of 1, 2, 4, 8 DAS5 nodes and 1, 4, 16 cores per node were tested, using the following command:

```
prun -np n -c -script $PRUN_ETC/prun-openmpi ./gol-par <M> <N> <T> <W> <C>
```

where n is the number of DAS5 nodes and c is the number of cores per node. As for the arguments, M is the world width (columns), N is the world height (rows), T is the number of timesteps, and W and C are flags for printing purposes, that were taken equal to 0, so that printing time was not included in the measurements. Each combination of nodes/cores was run three times and the average of the results was taken. The results are given in three graph categories for each grid case, one for each type of metric, with the number of processes on the X axis and the measurements on the Y axis.

In **Figures 1-3** the metrics for the small grid are given. One can see that the results are not ideal. There is a decrease in measured times due to parallelization, but speedups are low, reaching only up to 5.82 for the maximum number of processes. This is due to the small problem size, which includes a small number of computations, and therefore leads to a low computation to communication ratio, especially as communication increases with the number of processes. Regarding the single process results, the process communicates the boundary conditions to itself, introducing unrequired communication, which lead to lower performance compared to the sequential program. An interesting fact is that in the cases of 4 and 8 processes, the set ups with 4 cores per node give better results. This is probably due to the fact the there are processes that share memory and the messages between them do not have to travel through the network. Another notable case is this of the single node with 16 cores, which does not only present a higher speedup than the 4x4 case, but it even outperforms the 4x8 and 2x16 setups, which have twice more processes. This result could also be due to the fact that in this case, all processes lie in the same node, sharing memory, so there is no need for any messages to use the network. In general, as the number of processes increases, the reduction in elapsed time and thus the increase in speedups decrease, with the efficiency dropping to really low levels, down to 4.5%. This fact leads to the conclusion that for a small problem size, using even more resources to

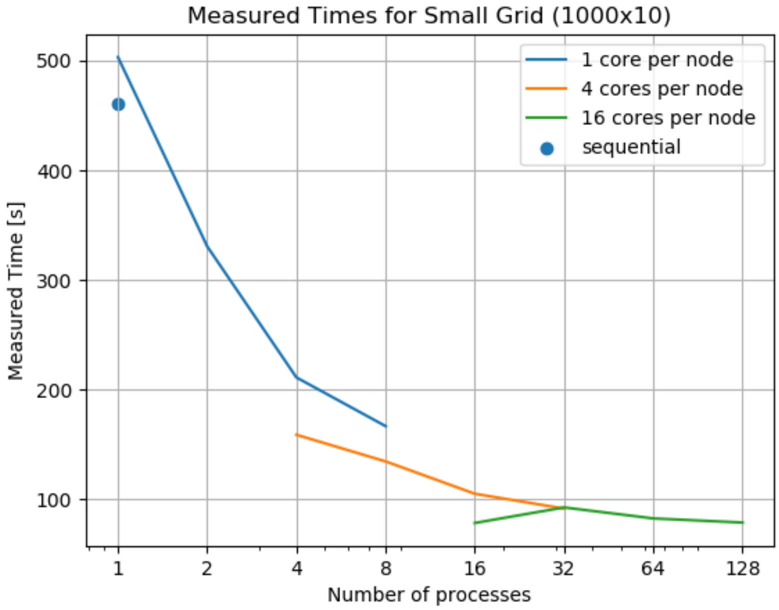

Figure 1. Measured times for a small 1000x10 grid (20 million timesteps) on 1, 2, 4, 8 DAS5 nodes and 1, 4, 16 cores per node.

parallelize the program will not be efficient, since after a level of parallelization, the communication overhead increases a lot. We, therefore, need to increase the problem size, in order to obtain better metrics. The results for the small grid are also given in **Table 1** below.

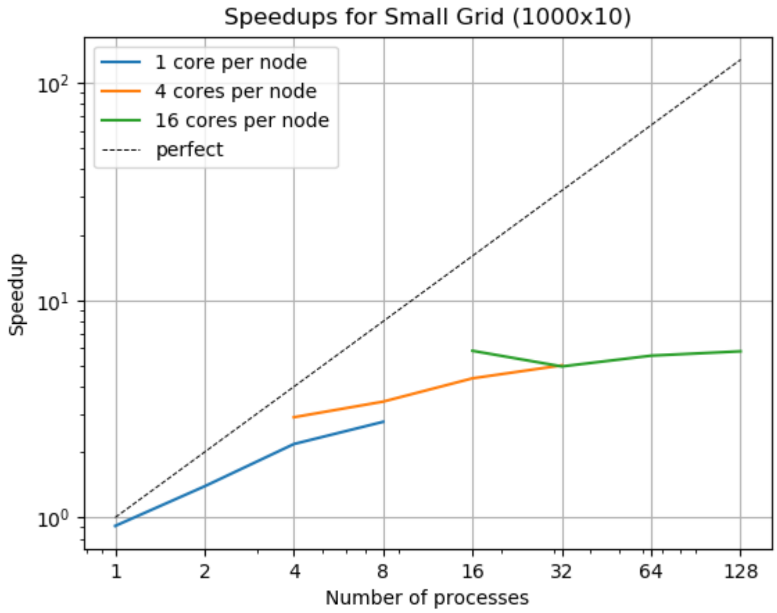

Figure 2. Speedups for a small 1000x10 grid (20 million timesteps) on 1, 2, 4, 8 DAS5 nodes and 1, 4, 16 cores per node.

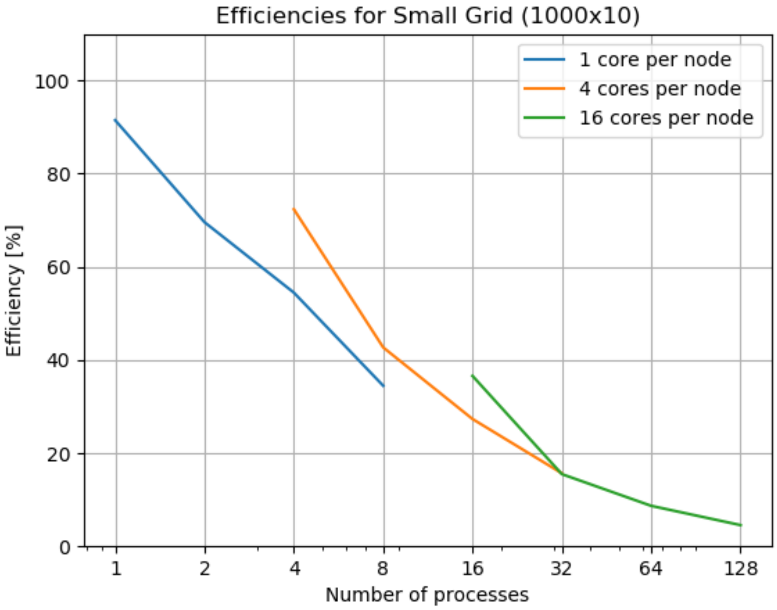

Figure 3. Efficiencies for a small 1000x10 grid (20 million timesteps) on 1, 2, 4, 8 DAS5 nodes and 1, 4, 16 cores per node.

Table 1. Metrics for the 1000x10 grid (20 million timesteps).

Processes (nodes x cores)	Measured Time (s)	Speedup	Efficiency (%)
Sequential	460.15	1	100
1 (1x1)	503.16	0.91	91.4
2 (2x1)	330.71	1.39	69.6
4 (4x1)	211.26	2.18	54.5
8 (8x1)	152.71	3.01	37.7
4 (1x4)	166.87	2.76	68.9
8 (2x4)	134.72	3.41	42.7
16 (4x4)	105.26	4.37	27.3
32 (8x4)	91.75	5.01	15.7
16 (1x16)	78.63	5.85	36.6
32 (2x16)	92.81	4.96	15.5
64 (4x16)	82.79	5.56	8.7
128 (8x16)	79.04	5.82	4.5

Moving on, in **Figures 4-7**, the results for the large grid are presented. It is evident that the increase in problem size leads to a significantly more efficient parallelization. This time, the measured time is almost halved each time we double the number of processes, with the speedups almost doubling and the efficiencies staying in high levels, over 70%, even with 128 processes. Also, the 1x1 parallel case is now much close to the sequential. This increase in performance is expected, since the number of computations is now much higher, allowing for high computation to communication ratios, even when the number of total cores increases a lot. Each process has to communicate 4 rows of 1000 cells, but the rows to compute are approximately 1000000/p, whereas, in the previous problem size, each process had to

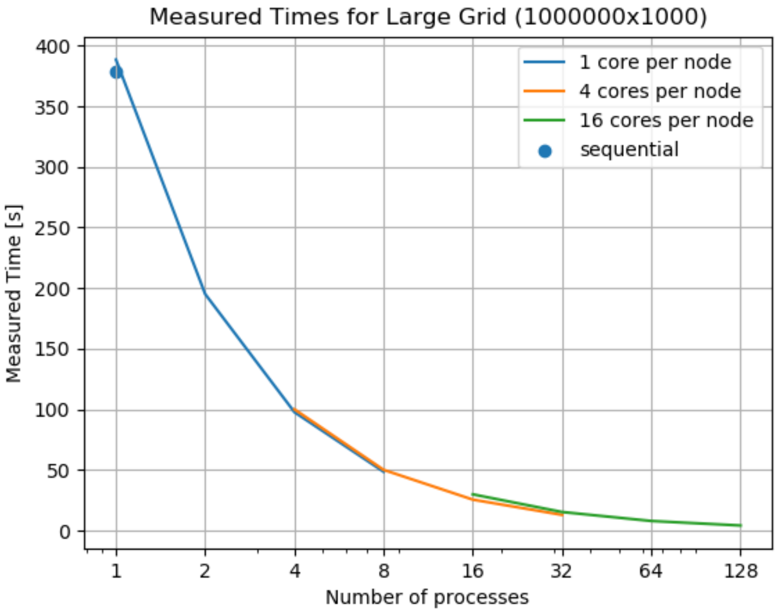

Figure 4. Measured times for a large 1000000x1000 grid (100 timesteps) on 1, 2, 4, 8 DAS5 nodes and 1, 4, 16 cores per node.

communicate 4 rows of 10 cells and compute approximately 1000/p rows, where p is the number of processes. Therefore, the computation/communication ratio is much higher now, explaining the notable increase in performance.

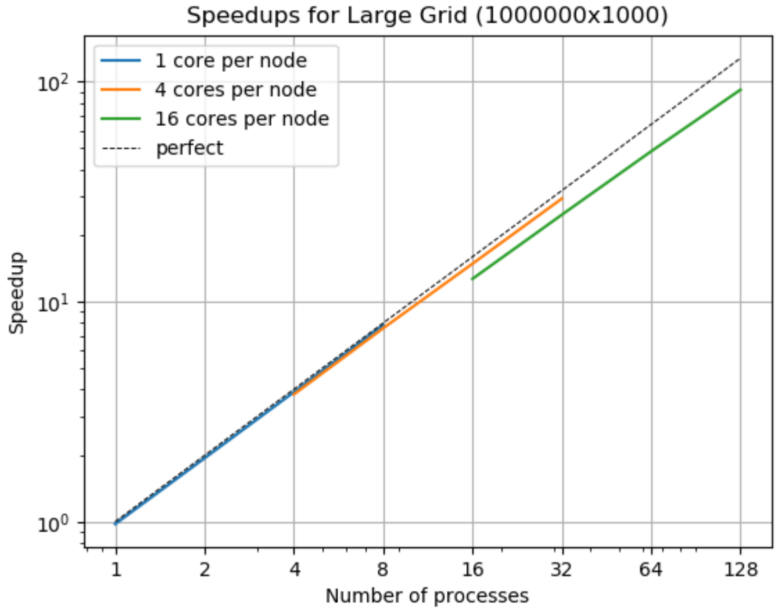

Figure 5. Speedups for a large 1000000x1000 grid (100 timesteps) on 1, 2, 4, 8 DAS5 nodes and 1, 4, 16 cores per node.

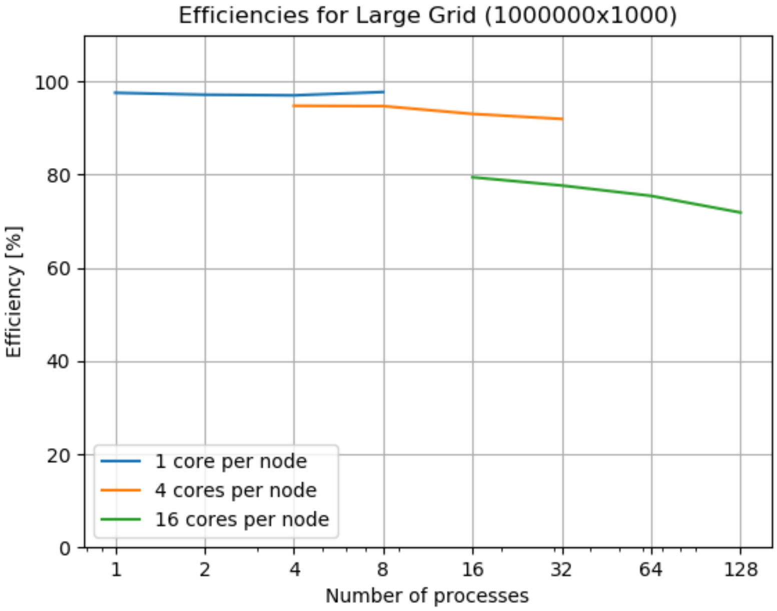

Figure 6. Efficiencies for a large 1000000x1000 grid (100 timesteps) on 1, 2, 4, 8 DAS5 nodes and 1, 4, 16 cores per node.

An interesting fact is that this time, the cases of equal processes but more cores per node seem to have lower performance, compared to the corresponding setups with more nodes and less cores per node, leading to the conclusion that large problem sizes are not favored by the memory sharing. This could be due the huge amount of data being all on the same processor and slowing it down, also explaining the general step change in efficiency from the orange to the green line. In general, for all setups up to 32 total cores, the slopes are steady, with almost perfect linear speedups and the decrease in efficiency being negligible. Then, at 64 and 128 processes, performance slightly worsens, with a tendency to start dropping more significantly at more than 128 total cores. The numbers for the abovementioned results are given in **Table 2**.

Table 2. Metrics for the 1000000x1000 grid (100 timesteps).

Processes (nodes x cores)	Measured Time (s)	Speedup	Efficiency (%)
Sequential	378.98	1	100
1 (1x1)	388.44	0.97	97.5
2 (2x1)	195.04	1.94	97.1
4 (4x1)	97.66	3.88	97
8 (8x1)	48.47	7.82	97.7
4 (1x4)	99.97	3.79	94.8
8 (2x4)	50.02	7.57	94.7
16 (4x4)	25.46	14.88	93
32 (8x4)	12.88	29.42	91.9
16 (1x16)	29.82	12.71	79.4
32 (2x16)	15.25	24.85	77.6
64 (4x16)	7.85	48.28	75.4
128 (8x16)	4.12	91.99	71.8

4. Bonus Work

The first bonus experiment (gol-parb1.c) is the comparison of the row wise grid decomposition of the basic implementation, with a column wise one. In order to split the board in blocks of complete columns, we have to create specific *MPI_Datatypes*, since the elements of a column do not occupy contiguous addresses in memory, as it was the case of the rows. The first datatype needed is an *MPI_Type_vector*, with a length equal to the whole world height and a stride, equal to the whole world width, so that the next vector element coincides with the next board column element. However, a modification to the standard vector datatype is required, using *MPI_Type_create_resized*, so that the next vector corresponds to the next board column. This datatype is used from process 0 in *MPI_Scatterv* to send the correct number of columns to the corresponding processes and in *MPI_Gatherv* to receive them back. Another resized vector datatype is required, with the only difference being that this time, the stride is equal to the subworld width of the current process, so that it coincides with the subworld columns. This datatype is used from each process in *MPI_Scatterv* to receive the correct number of columns and in *MPI_Gatherv* to send them back to the root process. Finally, an unmodified vector datatype of stride equal to the subworld width is used to communicate the boundary columns to the neighbouring processes. The performance of this implementation is strongly dependent on the grid shape. As mentioned earlier, the row wise decomposition shines in cases of grids with more rows than columns. On the other hand, the column wise splitting is more efficient in simulations of worlds with larger width than height, for the same reason that the computation/communication ratio is higher. The two implementations are compared for small and large grids of both shapes. The first comparison is between an 1000x10 grid and a 10x1000 grid, for 10 million steps. Those grids are only compared for up to 8 total cores, so that each process has at least 1 column of data and their metrics are presented in **Tables 3** and **4**, as well as in the set of graphs of **Figure 7**. It is evident that the 1000x10 grid, which has more rows than columns, favors the row wise decomposition, proving that our theoretical expectations were correct. In fact, the difference between the two approach is huge, with the performance of the column wise decomposition approach being unacceptably low. The metrics of the parallel algorithm are even worse than the sequential one, with slowdowns in all parallel setups. This is caused by the high amount of communication introduced by the column wise decomposition, since each process has 1 or 2 huge columns that they need to communicate with neighbours. Especially for processes that have 1 column, performance is expected to be the worst, since they have to communicate that column twice (to both neighbours), leading to much more communication than computation. On the other hand, the processes of the row wise approach are responsible for many short rows, which results in a much higher computation to communication ratio. Regarding the 10x1000 grid, in this case, since there are more columns than rows, the column wise decomposition is more efficient. However, the row wise approach is as bad as the column wise in the previous grid. Only the 2x1 setup presents a higher measured time than the sequential program, which is again due to the introduction of a high amount of communication with the parallelization of the algorithm, with the reduction in computation time not being enough to compensate for this loss. However, in the next two setups, it outperforms the sequential program. This fact leads to the conclusion that the use of the earlier mentioned vector *MPI_Datatype,* in order to split the world by columns, is highly inefficient if used in wrong grid shapes.

Table 3. Metrics of both approaches for the 1000x10 grid (10 million timesteps).

Processes (nodes x cores)	Row Wise Time (s)	Column Wise Times (s)	Row Wise Speedup	Column Wise Speedup	Row Wise Efficiency	Column Wise Efficiency
Sequential	230.01	230.01	1	1	100	100
2 (2x1)	166.95	514.34	1.37	0.44	68.9	22.3
4 (4x1)	105.8	462.49	2.17	0.49	54.3	12.4
8 (8x1)	76.36	428.38	3	0.53	37.6	6.7

Table 4. Metrics of both approaches for the 10x1000 grid (10 million timesteps).

Processes (nodes x cores)	Row Wise Times (s)	Column Wise Time (s)	Row Wise Speedup	Column Wise Speedup	Row Wise Efficiency	Column Wise Efficiency
Sequential	233.24	233.24	1	1	100	100
2 (2x1)	253.11	166.7	0.92	1.4	46	69.9
4 (4x1)	205.49	109.26	1.13	2.13	28.3	53.3
8 (8x1)	180.78	82.04	1.29	2.84	16.1	35.5

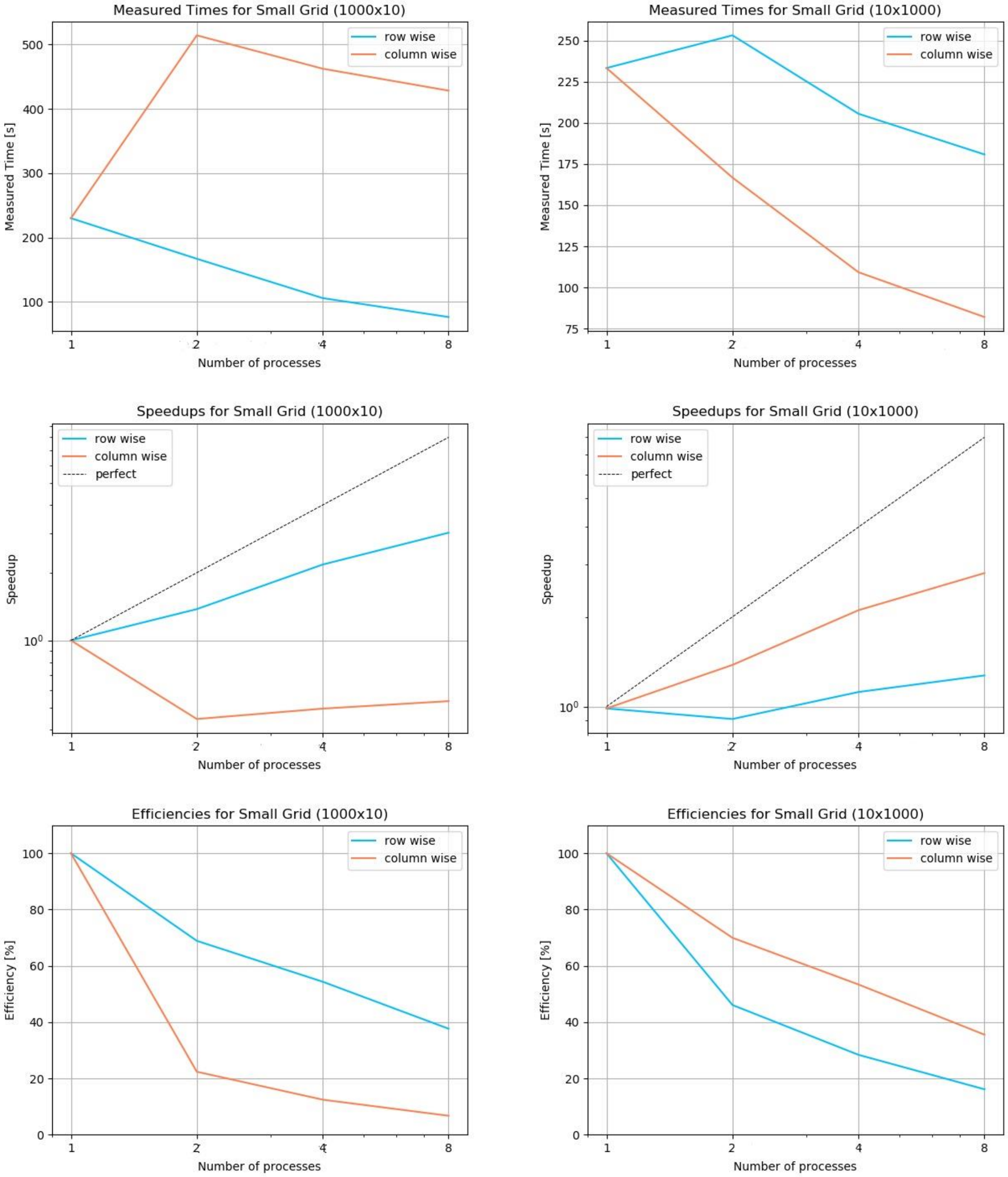

Figure 7. Comparison between the row wise and the column wise decomposition for a small problem size and 10 million timesteps. The metrics for the 1000x10 grid are on the left and for the 10x1000 grid are on the right side. Measured times are on top, speedups in the middle and efficiencies at the bottom.

Moving to larger problem sizes, the same comparison is presented in the plots of **Figure 8**, for an 1000000x1000 and an 1000x1000000 world, for 100 steps. The general behavior is similar to the small grid cases, however there is a notable difference. This time, in the case of the "wide" grid, even though the column wise decomposition is still the most efficient, it only outperforms the row wise one by little, with the difference only being notable at the 16 cores per node setups. On the other hand, the column wise approach is still notably inefficient in the case of the "high" 100000x1000 grid. When the number of processes gets higher than 16, the column wise approach presents a stabilization inefficiency around 30%, while the corresponding row wise efficiency is over 70%. This huge difference is probably again due to the vector *MPI_Datatype* used in this approach, supporting the opinion that it should be strictly used in grid shapes that favor the column wise decomposition. However, the column wise approach seems to increase its performance at more processes. The numbers for the discussed metrics are given in **Tables 5** and **6**.

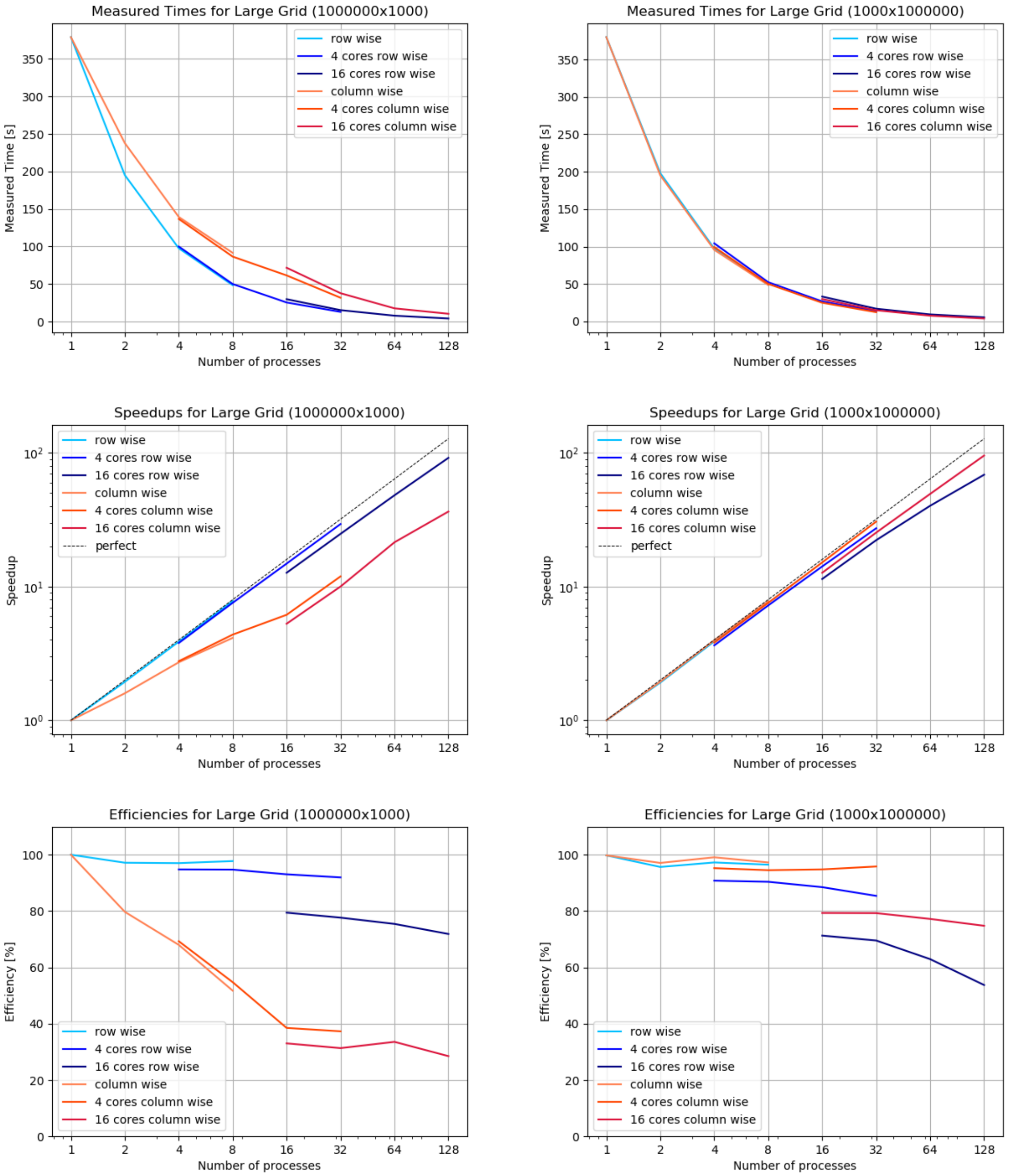

Figure 8. Comparison between the row wise and the column wise decomposition for a large problem size and 100 timesteps. The metrics for the 1000000x1000 grid are on the left and for the 1000x1000000 grid are on the right side. Measured times are on top, speedups in the middle and efficiencies at the bottom.

Table 5. Metrics of both approaches for the 1000000x1000 grid (100 timesteps).

Processes (nodes x cores)	Row Wise Measured Time (s)	Column Wise Measured Time (s)	Row Wise Speedup	Column Wise Speedup	Row Wise Efficiency (%)	Column Wise Efficiency (%)
Sequential	378.98	378.98	1	1	100	100
2 (2x1)	195.04	237.71	1.94	1.59	97.1	79.7
4 (4x1)	97.66	139.36	3.88	2.71	97	67.9
8 (8x1)	48.47	91.56	7.82	4.13	97.7	51.7
4 (1x4)	99.97	136.75	3.79	2.77	94.8	69.2
8 (2x4)	50.02	86.53	7.57	4.37	94.7	54.7
16 (4x4)	25.46	61.51	14.88	6.16	93	38.5
32 (8x4)	12.88	31.75	29.42	11.93	91.9	37.3
16 (1x16)	29.82	71.72	12.71	5.28	79.4	33
32 (2x16)	15.25	37.79	24.85	10.02	77.6	31.3
64 (4x16)	7.85	17.64	48.28	21.48	75.4	33.5
128 (8x16)	4.12	10.39	91.99	36.47	71.8	28.5

Table 6. Metrics of both approaches for the 1000x1000000 grid (100 timesteps).

Processes (nodes x cores)	Row Wise Measured Time (s)	Column Wise Measured Time (s)	Row Wise Speedup	Column Wise Speedup	Row Wise Efficiency (%)	Column Wise Efficiency (%)
Sequential	379.85	379.85	1	1	100	100
2 (2x1)	198.11	195.19	1.91	1.94	95.8	97.3
4 (4x1)	97.42	95.62	3.89	3.97	97.4	99.3
8 (8x1)	49.11	48.71	7.73	7.79	96.6	97.4
4 (1x4)	104.36	99.49	3.63	3.81	90.9	95.4
8 (2x4)	52.41	50.13	7.24	7.57	90.5	94.7
16 (4x4)	26.77	24.99	14.18	15.2	88.6	95
32 (8x4)	13.87	12.36	27.38	30.73	85.5	96
16 (1x16)	33.23	29.86	11.43	12.72	71.4	79.5
32 (2x16)	17.03	14.94	22.3	25.42	69.7	79.4
64 (4x16)	9.41	7.67	40.36	49.52	63	77.3
128 (8x16)	5.51	3.96	68.9	95.92	53.8	74.9

The second bonus experiment (gol-parb2.c) includes a more detailed analysis of the differences between blocking and non-blocking communication in the current application. As mentioned earlier, both approaches were tested, with the blocking one presenting the highest performance in most of the setups, except from tests on a small grid with 8 or more processes. In order to acquire a clearer understanding of this behavior, the elapsed time of communication and the elapsed time of computation is measured separately for the non-blocking approach. The *MPI_Wtime* operation is used around communication primitives, as well as computation blocks.

At the end of the simulation, apart from the total time, the computation and communication time are printed too. It should be noted that only the extra communication time is printed, and not the part of communication that is overlapped with computation. The results of some tests including cases in both grids with 4, 8, 16, and 128 are presented in **Table 7**.

Table 7. Computation, Communication and Total Measured times for different tests of the non-blocking approach.

Processes (nodes x cores), grid size, steps	Computation Time (s)	Communication Time (s)	Total Measured Time (s)
4 (4x1), small grid	218.76	23.3	249.27
8 (8x1), small grid	109.36	21.07	134.26
16 (4x4), small grid	56.9	25.35	84.75
128(8x16), small grid	9.95	28.05	39.2
4 (4x1), large grid	150.3	0.95	152.18
8 (8x1), large grid	75.16	0.005	75.37
16 (4x4), large grid	38.66	0.003	38.76
128 (8x16), large grid	5.73	0.19	5.98

Overlapping communication with computation leads to the same computations taking more time, compared to the time they required in the blocking approach, leading to longer total measured times. This could be due to the simultaneous work on two different tasks slowing down the process. Therefore, one can conclude that, in order for the non-blocking communication to worth the slowdown of computations, there must also be a significant amount of communication. This seems to be the case in the small grid simulations, where the number of required computations is low, and, as the number of processes grows, the amount of message passing becomes competent, and even higher at more than 16 processes. Therefore, decreasing the effective communication time by overlapping computation with communication results to notable gains in performance. This fact coincides with Amdahl's Law, which states that for a fixed problem size, increasing the degree of parallelization will never reduce the measured time to less than the part of the algorithm that cannot be parallelized. In this case, the communication between the processes could be described as the part that cannot be parallelized, and thus, after the parallelization degree limit is reached for blocking communication, we can use non-blocking to take advantage of the overlapping. On the other hand, when the amount of communication is low compared to the amount of computation, non-blocking communication is inefficient, since increasing the already high amount of computation, even more, leads to overall losses. This is the case in the large grid simulations, where the numbers of **Table 7** show that communication is complete while the computations are running. However, the overlapping highly increases the computation time, resulting in longer total measured times. It is expected though, that for a higher number of processes, non-blocking communication will be the most efficient for this problem size too, as it tends to get closer to the blocking's performance as the number of total cores increases. However, as the problem size increases, the blocking approach can continue being efficient for even higher degrees of parallelization, in agreement with Gustafson's Law.

5. Conclusions

The parallel game of life algorithm of the current study leads to a number of conclusions, regarding the efficiency of the different approaches, problem sizes, and number of processes. To begin with, it is evident that a fixed problem size can only be efficiently parallelized to a specific degree. After a number of processes, performance starts diminishing, and can even get worse compared to the performance of a setup with less total cores. Therefore, we need to move to bigger problem sizes in order to reach higher performance as the degree of parallelization increases.

Additionally, a high computation to communication ratio is a requirement for efficient parallelization. Grids with more rows than columns favor a row wise decomposition, whereas "wide" grids with more columns favor a column wise splitting, so that each process has more data to compute and less to communicate with its neighbours. A "wrong" choice of grid decomposition will most surely result in bad performance and may even present slowdowns. However, it seems that as the problem size grows, the inefficiency of a "wrong" decomposition diminishes for a fixed number of processes. However, as the number of total cores grows, the difference between the two decompositions increases.

Finally, as far as the type of communication is concerned, our simulations lead us to the conclusion that, for a fixed problem size, blocking communication is the most efficient until a specific number of total cores is reached. When that limit is reached, and the algorithm cannot benefit of higher parallelization, it is time to think of non-blocking message passing, in order to overlap communication with computation. However, as the problem size increases, blocking communication results in the highest speedups and efficiencies in the current application, even for higher degrees of parallelization.