Conway's Game of Life: a parallel implementation on MPI

Sokratis Anagnostopoulos, CID: 00986040, Imperial College London, 2020
github repository: https://github.com/acse-2019/acse-6-individual-assignment-acse-sa1619

Abstract

This work aims at implementing an efficient parallel algorithm for Conway's Game of Life using MPI (distributed memory protocol C++ library). Several techniques have been implemented towards improvements in performance, robustness as well as the generalisation of the code for more complex processor grid configurations or other applications. More specifically, these techniques include attempts for optimal, as well as efficient, processor grid configurations, distribution of computational grid into cores, treatment of special cases that arise when using periodic boundary condition and compact definition of Data types for the communication between the processors' boundaries. Several example animations have been created using a separate post-processing code, demonstrating the functionality of the implemented methods. The performance of the algorithm is also analysed using Imperial College High Performance Computing resources (HPC), in order to access its efficiency and speedup ratio measures. The results show that the Speedup ratio and the Parallel efficiency indices follow the Amdahl's Law only for coarse domain decomposition into a relatively small number of cores, where the data transferring time is negligible compared to the calculation time in each core. When more cores are deployed, the efficiency of the algorithm is reduced, depending on the problem size and cores number, and it is found to agree well with an approximate theoretical expression, after appropriate adjustment of problem-specific coefficient k, related to the relative cost of communication and computation.

1. Algorithm features

1.1 Intro

The cellular automaton is a simulation technique that can be used to model many physical mechanisms and phenomena, in different areas, like the tumor growth, the interaction of species, the chemical reactions, etc. The Game of Life is a remarkable example of the application of this technique, created by John Conway [1].

1.2 Grid division

The processor grid is automatically divided in the best possible configuration, considering only cases where no processor communicates with more than one neighbour processor through the same boundary. That is, for any non-prime processor number, the two factors that are closest to each other will be used, whereas for prime numbers the processor grid always has 1 row. Indicative examples are shown in Figure 1, where a 11 by 11 initial grid is allocated to 9 (a) and 11 (b) processors. In Figure 1a, 9 processors are divided in a 3 x 3 and in Figure 1b, 7 processors are divided in a 1 X 7 grid. As for the distribution of the initial simulation grid into each core, in the first case (Fig. 1a), since the remainder of the division 11/9 is 2, each of the first two processors take one extra row and column. Thus, the lower right processor (id = N) always has either 1 less, or equal number of rows/columns to the upper left processor (id = 0). For the second case (Fig. 1b), the number of the used processors is prime and so each core takes a horizontal strip composed of either 2 or 1 columns of the initial grid, by following the same procedure where the first few cores share the remainder of the division.

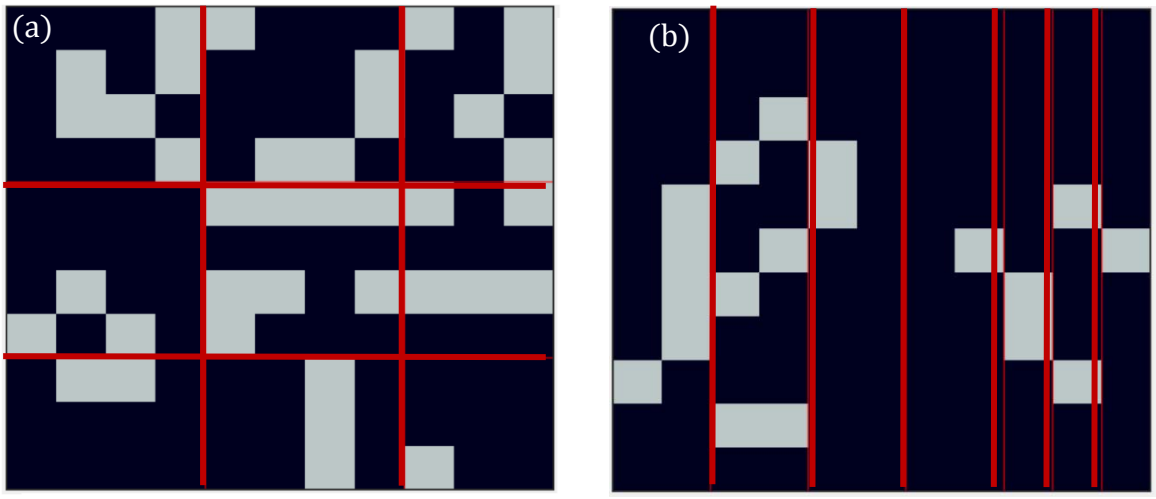

Figure 1. *Two snapshots of a 11 x 11 simulation with random-generated initial condition, showing the processor configuration using 9 cores (a) and 7 cores (b).*

1.3 Data types

An important feature of the presented algorithm is the use of special Data types which bundle all memory positions in one contiguous message [1] to be sent across the boundaries of each core. For the examined problem, since each core communicates with 8 neighbours, it should have 16 distinct Data types (8 for sending and 8 for receiving). Thus, at each core's initialisation, these types are created only once, so that they can later be used for the communication. Note that corners are always composed of 1 cell while sides are composed of more cells. Thus, a simple method for the creation of the Data types would be to type 16 distinct blocks of code to cover all cases, which however is time consuming but most importantly cannot be generalised for cases where one core has more than 8 neighbours, in which even more Data types would be required for the communication. Instead, a more efficient approach adopted by the presented algorithm is to first deal with the corners and then with the sides, so that only two for-loops are necessary for all the Data types' creation. As shown in Figure 2a, always starting from the upper left corner and rotating clock-wise, Data types 0 – 7 represent the corner sends and receives. Note that by adding 4 to the sending-types (in black) the corresponding corner receiving-types (in red) are obtained. Similarly, Data types 8 – 15 represent the side-sending and receiving types. The final product of this technique is the 1D vector, *"type_map"*, which is later used in the communication section so that the correct calling sequence of the appropriate Data types is maintained. For example, knowing that MPI_Isend and MPI_Irecv are called together in the same loop and that the vector which stores all Data types from 0 to 16 is called *"data_types"*, then their corresponding Data types for the upper left corner will be:

data_types[type_map[0]] and *data_types[type_map[0] + 4]*, respectively.

Note that the central *"0"* of the 2D array is neglected since the algorithm skips the communication between the processor and itself for this scenario.

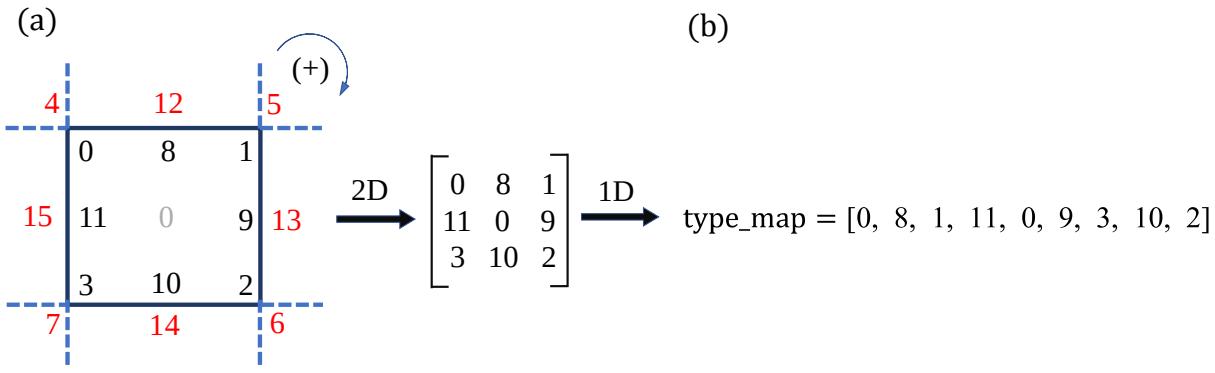

Figure 2. *Data types required for every core (a), where black numbers denote sending and red receiving types while the produced 1D vector maintains the correct clock-wise sequence to be used in the communication section of the algorithm (b).*

Considering the above Data type sequencing, provides a compact way to create these contiguous structures, which can easily be applied for more general cases where each core has an arbitrary number of neighbours. Note that for such a generalisation two simple additions should be made in a future version: a) a simple sub-routine that finds any processor corners coordinates to precisely define the start and end points of each Data type and b) that the loop which finds all side Data types should be divided into 4 loops (since each one of the 4 sides could be linked to a different number of core neighbours).

The above technique covers most of the core grid configurations. However, several special cases arise which also need to be considered for a holistic approach. These cases are divided into two sub-categories: a) the 2 X N processor grid and b) the prime number grid processor cases. For both of the above scenarios, the need of unique tag number that corresponds to each unique communication type arises, so that all communications can be distinct.

1.4 Scenario 1: 2 X N core-grid

A simple 2 X 2 example is shown in Figure 4a, where core number 0 has 4 distinct communications with core number 3, each of which is defined by a different Data type. Therefore, an additional set of 1D mapping vectors must be employed, *"send tag_num"* and *"receive tag_num"*, so that all the communication Data types are linked with a unique send-receive tag number couple. For example, sending from the upper-left corner and receiving from the bottom-right corner have the same tag number equal to 0. The same logic applies to all other corners and sides. Again, note that the central *"0"* of the 2D array is neglected since the algorithm skips the communication between the processor and itself for this scenario.

1.5 Scenario 2: Prime number core-grid

Similarly, when the core number is prime, having a single row configuration allows for two different communications between a core and itself to occur, as shown in Figure 4b. For that case, a new set of tag number vectors needs to be considered instead, which will replace the standard set previously mentioned only when the number of rows is equal to 1. The corresponding tag numbers for sending and receiving are shown in Figure 6, and are derived with the same methodology.

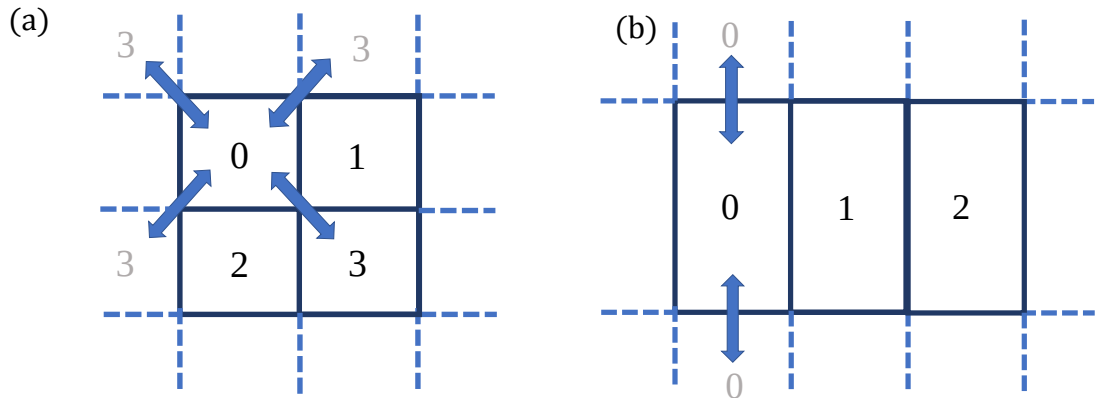

Figure 4. *Simple 2 X 2 (a) and 1 X 3 (b) examples, showing how the same communication between processors can be repeated but for different Data types each time.*

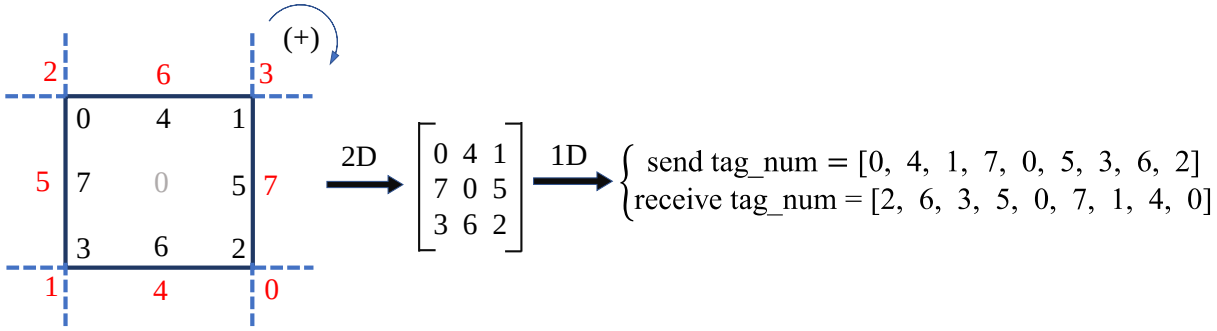

Figure 5. *Unique tag numbers that link diametrically opposed processor corners or sides, to cover almost all periodic boundary condition cases, including 2 X N cases.*

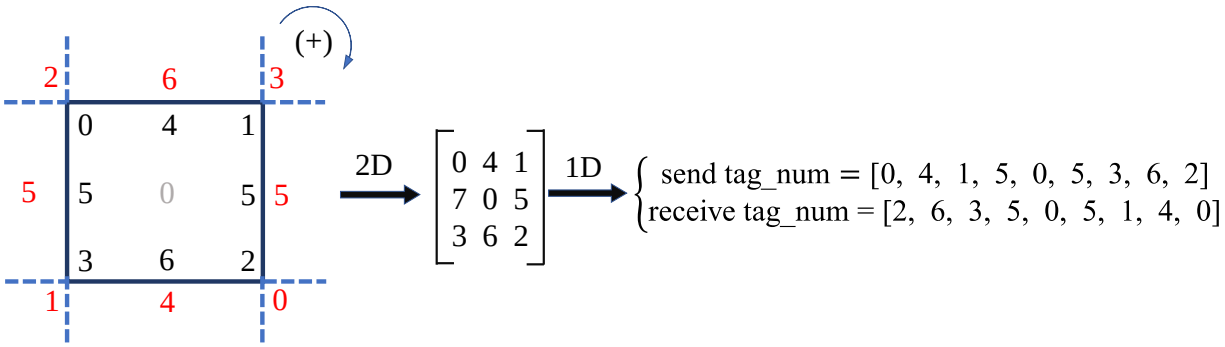

Figure 6. *Unique tag numbers that link diametrically opposed processor corners or sides, to cover almost periodic boundary condition cases where the number of cores is prime.*

1.6 Post-processing

A demonstration of the working algorithm is shown in the attached video files 1-7 and their description is included in the Readme file. The post-processing algorithm was written in MATLAB, mainly due to the integrated libraries that facilitate the handling of unequal sized arrays but also due to the live animation feature that facilitated the debugging procedure. Nevertheless, the produced text files can easily be post-processed by any other available programming language.

2. Results and performance discussion

The first Figure (Fig. 7a) shows the variation of total computation time as function of the number of cores used, for an indicative mesh of 1000 x 1000 pixels. The reduction rate of total time is high at first, but as more cores are deployed, the rate reduces, reaching to a minimum total time for about 140-150 cores, and slightly increasing for even more cores. This inefficiency is expected, due to the increasing share of time spent for communication compared to the calculation time, as the same computational domain is divided in more and more cores (the domain portion calculated in each core reduces, whereas the overall communication time increases). The number of cores that minimizes the total time increases with the domain size.

Figure 7b is a detailed view of Fig. 7a for 6 to 30 cores that demonstrates the performance of the domain division methodology used in the algorithm. The results exhibit some scattering, because the computational time is too small to smooth out any small variations in cores performance (e.g. due to small temperature variation of the CPU). However, it is clear that the

time spent for the cases of prime cores number (7, 13, 17, 19, 23, 29) shows considerable time peaks than the rest, due the less efficient division of the domain into vertical strips in terms of the cumulative length of interfaces between cores.

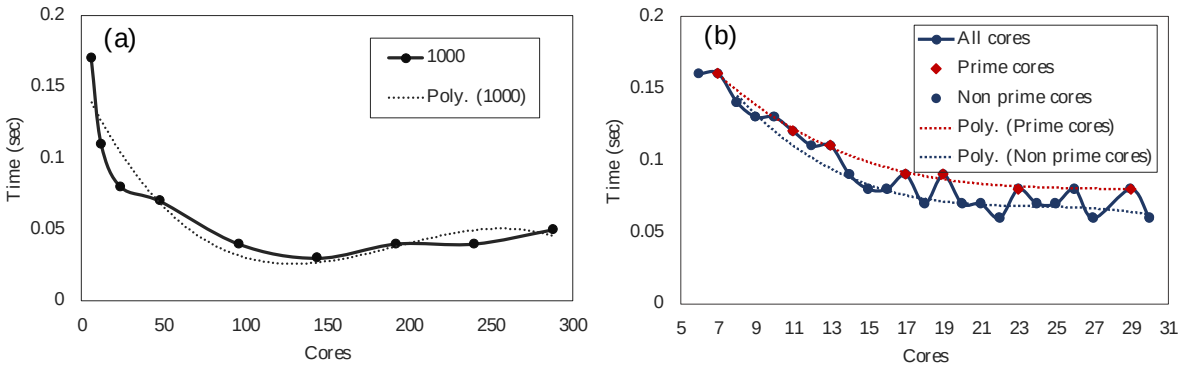

Figure 7. *Core number vs computation time of a 1000 by 1000 grid up to 288 (a) and 30 individual processors (b).*

The relation of total time with the number of cores is shown in Fig. 8a for various domain sizes. The log–log plots reveal that all curves have almost the same slope, at least at their first part (before the communication time becomes substantial and comparable to the calculation time). The initial slope is about 1:1, namely the required time is inversely proportional to the number of cores, as expected, but it is being reduced as the number of cores increases above 20.

The total time increases with the size of the domain (pixel count), and this correlation is quantified in the curves of Figure 8b, created for various core numbers. Here again, all curves exhibit similar slope, but this time at their right part, namely for larger domains, where again the data transferring time is very small compared to the calculation time of each core. This slope is now about 2:1, namely the total time increases with the square of the mesh size, or proportionally to the total number of pixels in the domain (domain size, P), which is reasonable.

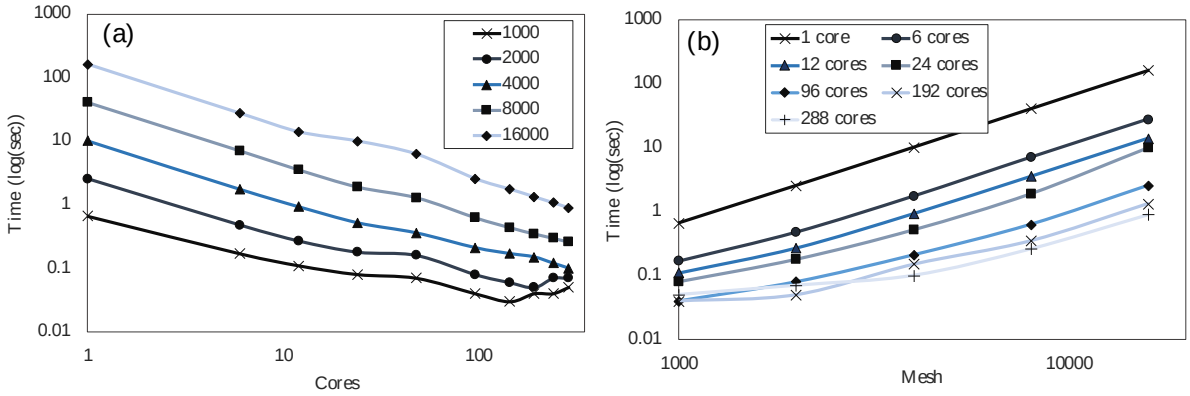

Figure 8. *Log-log plot of cores vs computation time for several dimensions (a) and grid dimensions vs computational time for different core numbers (b).*

The next two graphs of Figure 9 (9a, 9b) present the performance of the algorithm in terms of Speedup ratio, S, and Parallel efficiency, E, indices of the parallel computer code developed in this project. As discussed above, almost the entire code is parallelised, hence the fraction, f, in Amdahl's Law is close to 1 [3]. In that case, Eqs. (1) and (2) of this law give:

$$S_{real} = \frac{1}{1-f+\frac{f}{N}} = N \quad (1)$$

$$E_{real} = \frac{1}{N(1-f)+f} = 1 \quad (2)$$

thus $S \approx N$, and $E \approx 1$, respectively, where N is the number of cores. The actual Speedup Ratio and Parallel Efficiency of the code are calculated from the following corresponding equations (3) and (4):

$$S = \frac{T_1}{T_N} \quad (3)$$

$$E = \frac{T_1}{T_N} = \frac{S}{N} \quad (4)$$

and these performance results for the various examined domains are compared in the diagrams of Fig. 9. The achieved speed up ratio is quite lower than the maximum theoretical of Amdahl's Law ($S \approx N$, Fig. 9a), and the parallel efficiency is well below the maximum $E \approx 1$ (Fig. 9b), except from the first part of the curves for a small number of 10 - 20 cores. For higher number of cores, the parallel efficiency exhibits a drastic drop, followed by a lower and smoother decrease rate.

Moreover, the performance of the code becomes even less efficient for smaller domain sizes. This behaviour is the result of the communication time that is spent in addition to the calculation time, the fraction of which in the total computation time becomes larger for smaller domain sizes or for greater core numbers. Therefore, the actual performance results of the parallel code should be compared with the more realistic estimation of the domain decomposition efficiency (taken from the lecture notes), which for 2D square domain becomes:

$$E_{2D} \approx \frac{1}{1+k\sqrt{(\frac{N}{V})}} \approx \frac{1}{1+k\frac{\sqrt{N}}{A}} \quad (5)$$

where A is the domain size in pixels ($A = V^{1/2}$ for a square domain).

The above expression contains the coefficient k, which is problem-specific. The Conway's Game of Life, as a cellular automaton, belongs to the Structured Grid Dwarf [4], that has specific calculation and communication patterns, where the relative portion of time for calculations and for transfer of data is specific. Consequently, the value of the parameter k should be constant for any domain decomposition and size and thus should take a fixed value for the present Game of Life problem.

This value is regulated to the value $k \approx 500$, so as to match the parallel efficiency results of an intermediate size domain of 4000 x 4000 pixels, as can be seen in Fig. 9d. Then the above Eq. (5) with this value of k is applied to other domain sizes, and the corresponding curves are also plotted in Fig. 9d, showing a satisfactory agreement with the code's real parallel efficiency of the code.

The same satisfactory agreement can be also observed in the Speedup Ratio results, which are compared in Fig. 9c, where the theoretical speedup ratio is computed as $S = E * N$, and the real S is calculated from Eq. (3).

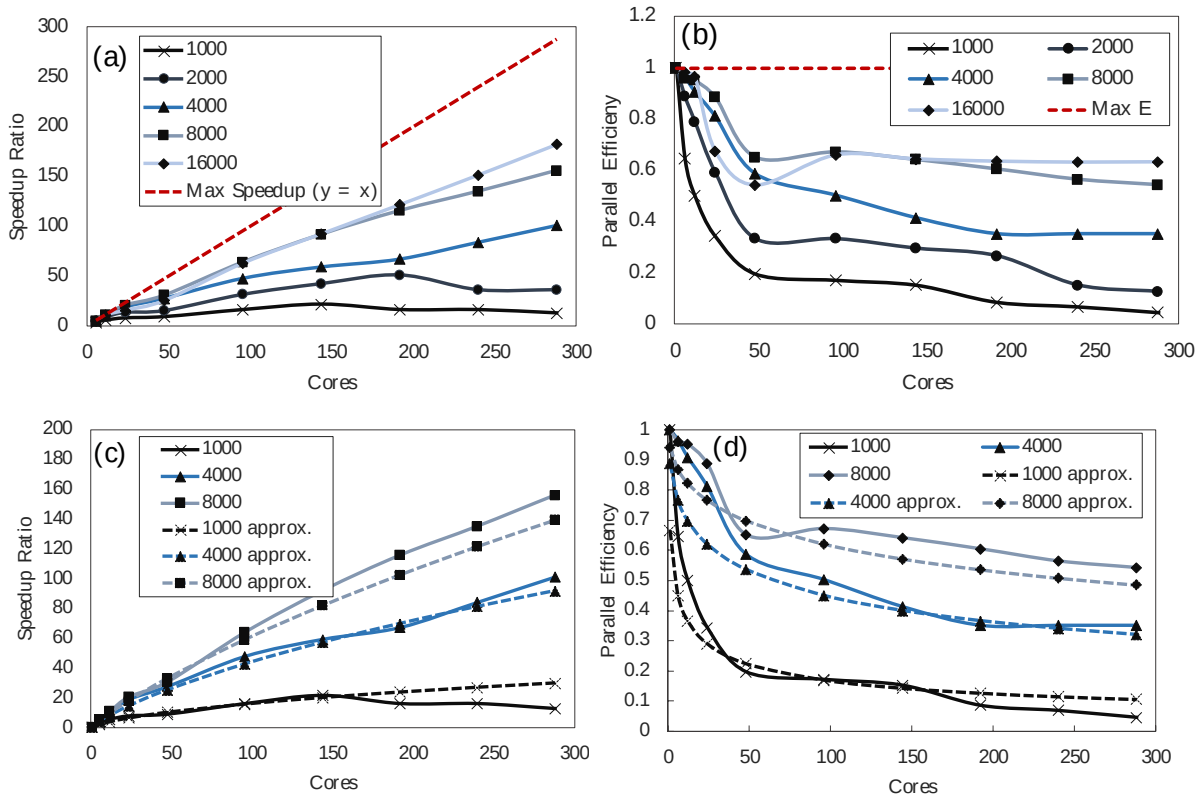

Figure 9. *Performance results of the parallel algorithm. Speedup Ratios (a), (c) and Parallel Efficiency (b), (d) for several computational grid dimensions.*

3. Conclusions

The performance of the parallel code developed in the present project to simulate the Game of Life problem is satisfactory and in agreement with the theoretically estimated, concerning both the Parallel efficiency and the Speedup ratio indices.

The structure of the code and its different components is compact, and its robustness has been demonstrated for several scenarios using a post-processing algorithm. The implementation of the data_type and tag_num arrays significantly increases the potential of the algorithms scalability. For that reason, this technique could be easily adapted to other domain decomposition patterns (e.g. arbitrary configurations with cores with more than 8 neighbours or triangular shapes), and hence the parallel code could be further generalised for applications in various different problems (e.g. CFD, FEM, etc). Finally, such generalization allows for adaptive domain decomposition, in cases that the calculation load is non-uniformly distributed within the domain or even if it is being varied during the solution, (e.g. in adaptive mesh refinement simulations).

References

[1] Gardner M. (1970), The fantastic combinations of John Conway's New Solitaire Game "Life", *Scientific American*, **223**, 120-123.

[2] Available at https://pages.tacc.utexas.edu/~eijkhout/pcse/html/mpi-data.html

[3] Rodgers D.P. (June 1985), "Improvements in multiprocessor system design". *ACM SIGARCH Computer Architecture News*. New York, NY, USA: ACM. **13** (3): 225–231.

[4] Asanovic K., Bodik R., Catanzaro B., Gebis J., Hysbands P., Keutzer K., Patterson D., Plishker W., Shalf J., Williams, S., and Yelick K. (2006), The Landscape of parallel computing research: A view from Berkeley. *University of California*, Technical Report No. UCB/EECS-2006-183.

[5] Pacheco P.S. (2011), An Introduction to Parallel Programming, *Elsevier Inc*.

[6] Barlas G. (2015), Multicore and GPU Programming. An integrated Approach, *Elsevier Inc*.