Performance Analysis of Parallel Game of Life in MPI

Bo-Chian Chen
Vrije Universiteit Amsterdam
b.chen4@student.vu.nl

I. Problem Definition

Design, implement, and analyze the parallel performance of a Game of Life (GOL) application on the DAS-5 cluster using MPI (Message Passing Interface). The focus is on understanding the impact of parallelization on speedup and efficiency, considering different node and core configurations.

II. Implementation

I employ a 1D block partition method to parallelize the GOL. Initially, all workers are equipped with the entire configuration of the GOL matrix. Subsequently, each worker is tasked with updating its designated submatrix and verifying if the submatrix exhibits a cycle with one of the two most recent history matrices. Upon detecting a cycle, the worker signals the others to halt and transmits the iteration index to the root worker. Consequently, if a worker accumulates stop signals from all workers, the root prints the execution results and terminates the program. The following will deliberate on the role of functions in the main iterations.

A. strips_boundary_exchange

In this function, workers facilitate the exchange of borders in an efficient manner through a predefined order of processing. To optimize communication, a structured approach is employed: odd-numbered workers initiate by sending their borders to the subsequent worker and receiving from the preceding one, while even-numbered workers commence by receiving from the preceding worker and subsequently sending their borders to the next one. This alternating pattern continues until the final step, where the first worker exchanges borders with the last worker..

B. world_check_cycles

This function primarily checks for cycles within each worker's submatrix in the GOL simulation. If a cycle is detected, it updates the iter_pin variable to record the iteration at which the current submatrix exhibits a cycle. Subsequently, workers send a stop signal with a value of 1 to inform others about the cycle and collect stop signals from all workers.

Additionally, the function determines the existence of a cycle in the entire configuration matrix when the sum of stop signals equals the total number of workers. In such cases, the iterations are halted, the results are printed, and the program terminates.

C. world_timestep

This function is responsible for computing the next generation of the world according to the rules of the GOL. The computation is organized into strips, with each MPI process dedicated to processing its assigned strip. This division of labor optimizes the parallelization of the time step calculation.

D. decompose_domain

This function orchestrates the division of world rows among MPI processes.

III. Challenges and Solutions

A. The Function of the Main Worker

Initially, my approach was influenced by the existing codebase, resulting in a less optimal strategy where I gathered updated submatrices from workers before conducting cycle detection. This approach significantly impacted the program's performance. Upon reassessment, I discovered that guidelines recommend excluding the time required for data distribution, gathering, and printing from the evaluation.

I realized that the desired outcome only necessitated verifying the correctness of the stopping iteration and the cell count, unrelated to presenting the entire Game of Life configuration. Consequently, I refactored the implementation to empower each worker to autonomously check for cycles and communicate the final result to the main worker only upon reaching the end of steps or encountering a cycle. This refinement led to a substantial increase in the speedup of the program.

B. Order in Border Exchange

Initially, I synchronized MPI_Send and MPI_Recv operations across all workers, assuming sufficient channels to mitigate potential blocking issues. However, upon scrutinizing the time allocated for border exchange on each worker, a consistent discrepancy in communication time surfaced.

Upon investigation, it became apparent that delays occurred as workers had to wait for others to complete all MPI_Send operations. To address this inefficiency, I reconfigured the execution order by dividing workers into odd and even groups based on their numbers. Odd-numbered workers initiated by sending borders to subsequent workers and receiving from preceding ones, while even-numbered workers started by receiving from the preceding worker and then sending borders to the next one. This alternating pattern ensured that each worker performed an MPI_Send operation in one direction followed

by a corresponding MPI_Recv operation from the counterpart. This adjustment significantly enhanced the efficiency of the parallelized execution.

IV. Execution Results

In this section, the methodology for measuring wall time and selecting the time will be delineated. Subsequently, the outcomes of all executed processes will be presented for comprehensive analysis.

A. Operating environment

The system I use to run and test the parallel applications is called Distributed ASCI Supercomputer 5 (DAS-5). It is a wide-area distributed cluster designed by the Advanced School for Computing and Imaging (ASCI).

Cluster	VU
Nodes	68
Type	dual 8-core, 2x hyperthreading
Speed	2.4 GHz
Memory	64 GB
Storage	128 TB
Node HDDs	2*4TB
Network	IB and GbE
Accelerators	16*TitanX+ Titan/GTX980/K20/K40

B. Wall time measurement

The primary focus of wall time measurement resides in the timing of iterations within the main function. In the execution of the parallel version, the initial action is the invocation of updates at each worker. Consequently, the timer commences its tracking at the onset of the border exchange and cycle checking function. This timing encapsulates the entire duration of communication, including the time consumed by root worker to collect data throughout all iterations. As a result, the wall time is comprehensively gauged, offering a holistic measurement of the execution process.

C. Selection of the time

When determining the serial version's execution time, four program runs are averaged to establish the baseline. This average serves as the reference for calculating parallel speedup.

For the parallel version, precision is prioritized through four meticulous runs. The results are averaged as the representative execution time. This approach ensures accurate assessment of parallel performance.

D. Selection of the size

I intend to choose two illustrative world sizes: a medium-sized configuration, ensuring sufficient data for each process, and a significantly larger one, where the grid size is adjusted to allow the sequential version to complete within a few minutes. However, considering constraints on Das5, sizes exceeding 32000 are impractical. For the larger size, I have opted for 29440, a multiple of 128 and taking around 108 sec. Consequently, the medium-sized configuration is set around half of this value, i.e., 16000.

E. Fixed size with m=n=16000, steps=32

1) Serial version result:

Run 1	31.138
Run 2	31.147
Run 3	31.172
Run 4	31.182
Average: 31.160	

2) Parallel version results:

# Workers	# Nodes	Exec time	Speedup	Efficiency
2	2	17.0265	1.83008	0.91504
4	2	8.7165	3.57482	0.89370
8	2	4.88875	6.37381	0.79672
16	4	2.69	11.58364	0.72397
32	8	1.78575	17.44925	0.54528
64	8	1.5415	20.21407	0.31584
128	8	1.86625	16.69658	0.13044

F. Fixed size with m=n=29440, steps=32

1) Serial version result:

Run 1	108.584
Run 2	108.446
Run 3	108.483
Run 4	108.509
Average: 108.505	

2) Parallel version results:

# Workers	# Nodes	Exec time	Speedup	Efficiency
2	2	58.10225	1.86747	0.93373
4	2	29.28825	3.70470	0.92617
8	2	16.0725	6.75093	0.84386
16	4	9.00175	12.05369	0.75335
32	8	5.92675	18.30756	0.57211
64	8	3.5355	30.68995	0.47953
128	8	3.486	31.12574	0.24316

V. Performance Analysis

A. Communication/computation ratio

To assess program performance, I quantify both communication and computation aspects, employing a 1D block partition method where each worker updates N^2/P items per iteration. Communication involves obtaining data from two neighbors, translating messages of size N/P per iteration. The communication/computation ratio asymptotically approaches $O(P/N)$.

B. Efficiency analysis

The communication/computation ratio $O(P/N)$ implies that, in this parallel framework, the addition of processors is expected to yield diminishing marginal utility linearly. To validate this theoretical premise against practical experience, I plot the derivative of the communication/computation ratio alongside efficiency. Figure 1 demonstrates a consistent trend, affirming alignment with the theoretical assumption.

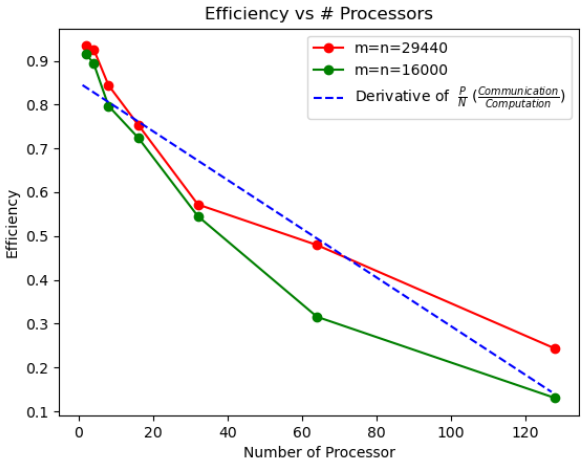

Fig. 1. Efficiency on sizes: m=n= 16000& 29440, steps=32

C. Speed up with both sizes

The speedup analysis, depicted in Figure 2 for varying numbers of workers, reveals an intriguing observation. Obviously, there is a flattening of the speed-up curve, which follows what we have discussed on the communication/computation ratio. Specifically, as assumed, the size 16000 exhibits diminishing marginal utility earlier, given its relatively smaller scale.

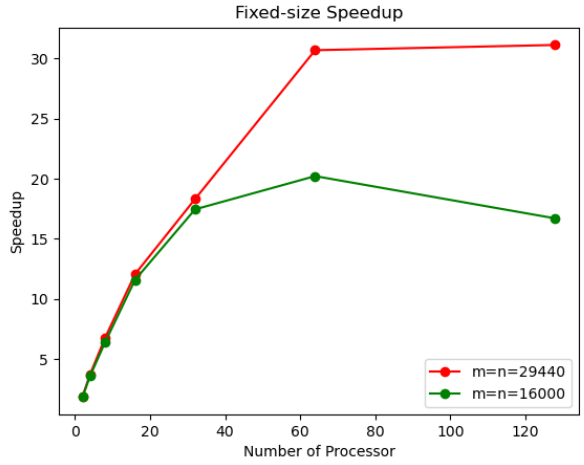

Fig. 2. Speed up on sizes: m=n= 16000 & 29440, steps=32

VI. Advanced Analysis

A. Same # cores with different # nodes

In this section, we delve into the nuanced analysis of parallel program execution, specifically focusing on scenarios where the program runs on the same core but with different pairs of nodes and varying numbers of workers per node. This exploration aims to unravel the intricacies of performance variations and uncover insights into the impact of node configuration on parallel processing efficiency.

B. Execution Results

m=n=16000, steps = 32		
# Workers per node	# Nodes	Exec time
16	1	3.1505
8	2	2.948
4	4	2.69
2	8	2.764
1	16	2.735

m=n=29440, steps = 32		
# Workers per node	# Nodes	Exec time
16	1	40.0995
8	2	8.933
4	4	8.5313
2	8	9.00175
1	16	9.1686

The performance analysis reveals a noteworthy trend illustrated in Figure 3. Optimal execution speed is consistently achieved when distributing cores uniformly across all nodes, while the slowest performance is consistently associated with concentrating all cores on a single node.

Of particular interest is the observation that employing all 16 cores on each node, especially with larger grid sizes, results in an impractically large execution time. Intriguingly, in such instances, the parallel version's execution time surpasses that of the serial version, as detailed in the accompanying table.

m=n=33280, steps=32			
Version	# Workers	# Nodes	Execution time
Serial	1	1	134.999
Parallel	16	1	288.601
Parallel	128	8	169.087

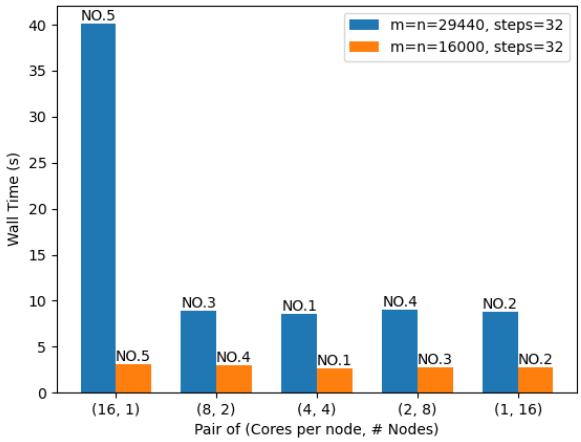

Fig. 3. Same # cores on different # nodes

VII. Latency Hiding

A. Concept

In this function, latency hiding is attempted through the use of asynchronous non-blocking MPI communication calls (MPI_Isend and MPI_Irecv). The asynchronous nature of these calls allows the program to continue with computation while waiting for communication to complete in the background. Thus, my idea is to implement new_world_timestep combining the function world_timestep and strips boundary exchange together, which helps in hiding the communication latency and potentially improving overall performance.

B. Implementation

1) Asynchronous Communication Calls: MPI_Isend and MPI_Irecv are used for communication. These are non-blocking versions of the standard MPI_Send and MPI_Recv functions, respectively. Non-blocking calls return immediately, allowing the program to proceed with other tasks while communication is in progress.

2) Interleave computation and communication order: The function new_world_timestep will start to communicate border information with others. At the same time, the function will execute the update of inner rows. That is, the rows that need border information will be updated lastly.

3) Computation as Waiting: While waiting for the communication to complete, the function performs local computations (one_line_update) on the matrix elements. These computations are performed for the inner rows in the submatrix and proceed independently of the ongoing communication.

C. Latency-hiding Execution Results (m=n=29440, steps=32)

# Workers	# Nodes	Exec time	Speedup	Efficiency
2	2	56.694	1.86747	0.93373
4	2	28.695	3.70470	0.92617
8	2	15.861	6.75093	0.84386
16	4	8.708	12.05369	0.75335
32	8	5.15	18.30756	0.57211
64	8	3.068	30.68995	0.47953
128	8	3.292	31.12574	0.24316

D. Latency-hiding Performance Analysis (m=n=29440, steps=32)

In Figure 4, the incorporation of latency hiding significantly enhances efficiency. Notably, a substantial improvement is observed with 32 and 64 processors, suggesting an optimal communication-to-computation ratio $O(P/N)$. This trend is evident in Figure 5 as well. Furthermore, with the auxiliary line of the derivative of communication/computation ratio, we can also notice that the one with latency hiding is more approach to it. This means that we implement it in a right manner since the derivative of communication/computation ratio means the theoretical trend of efficiency decreasing which is linear.

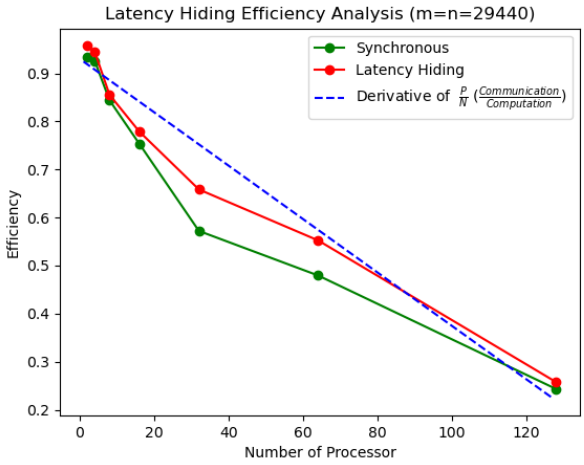

Fig. 4. Latency Hiding Efficiency Analysis (m=n=29440, steps=32)

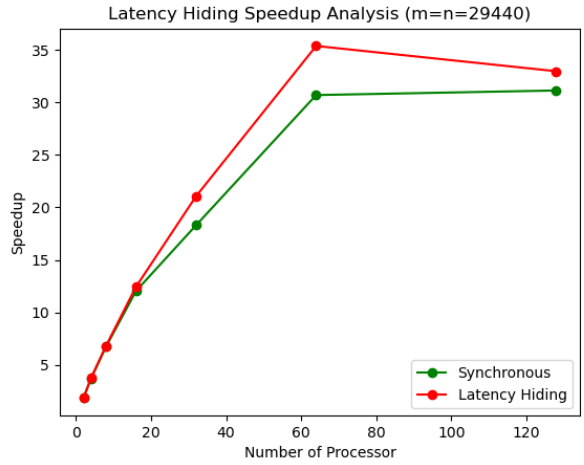

Fig. 5. Latency Hiding Speedup Analysis (m=n=29440, steps=32)

VIII. Overhead Analysis

A. Proportion benefits from parallel

m=n=16000, steps=32				
Version	# Workers	# Nodes	p. time	Exec time
Serial	1	1	31.164361	31.164372

In this equation: p. time is the time consumed by the proportion can benefits from parallel. As a result, f is equal to 0.99. (f is the fraction of the computation that must be executed parallel.)

B. Amdahl's Law

$$S = \frac{1}{1 - f + \frac{f}{P}} \tag{1}$$

In this equation: S is the theoretical speedup achieved by parallelizing a computation. f is the fraction of the computation

that must be executed parallel. And P is the number of processors used in parallel execution. With the result of f=0.99, by Amdahl's Law, our theoretical speed-up is 13.91 with four processors, P=16.

C. Gustafson's Law

$$E(P) = P + (1 - P) \times s$$

In this formula: E(P) represents the theoretical parallel speedup, P is the number of processors and s is the fraction of the computation that must be executed serially. With the result of f, s = 1-f = 0.01. By Gustafson's Law, our theoretical speed-up is 15.85 with four processors, P=16.

D. Computation time

To quantify the computation time, it involves measuring the duration dedicated to local computation tasks. Specifically, we assess the time taken for tasks such as matrix boundary wrapping, executing Game of Life rules to update the entire configuration, and verifying the presence of any cycles. As a result, the computation time is measured by evaluating the time spent within the functions world_check_cycles, world_border_wrap, and world_timestep.

E. Synchronization time

In the context of synchronization time measurement, the primary focus lies on assessing the MPI_Barrier at each iteration. This particular MPI function plays a pivotal role in synchronizing the distributed processes after the completion of each iteration.

Additionally, the timing analysis incorporates the measurement of time spent within the strips_boundary_exchange function. Given that each worker engages in multiple communication tasks, the intervals between successive communications serve as implicit synchronization points. Consequently, the timing mechanism initiates immediately after the worker completes the reception of data. The timing interval concludes upon the commencement of the subsequent communication, characterized by the initiation of data transmission.

This comprehensive approach aims to capture and quantify the synchronization overhead incurred during inter-process communication within the parallel program.

F. Communication time

In evaluating communication time, the primary metric involves measuring the time allocated to strips_boundary_exchange. Additionally, the MPI_Allreduce is timed within each iteration to capture its contribution. The computed communication time is derived by subtracting the accumulated synchronization time from the overall timing results.

G. Computation, Communication and Synchronization times

16 Workers on 4 Nodes with m=n=16000, steps=32	
Synchronization time	0.09246
Communication time	0.275
Computation time	2.499
Execution time	2.959

At the table above. the synchronization/communication ratio is 0.336, which is a little bit high. On the other hand, the communication/computation ratio is 0.11, which is in a acceptible range. Besides, it's interesting that this result also prove my experience in latency hiding. The time saved in latency hiding is 9.001 - 8.708 = 0.293, which is really close to the communication time 0.257 measured in this experience.

H. Analysis

# Workers	# Nodes	Sync time	Comm time	Comp time
2	2	0.000211	0.07	16.89
4	2	0.0297973	0.106	8.57
8	2	0.1157873	0.457	4.632
16	4	0.09246	0.275	2.499
32	8	0.1832368	0.16	1.442
64	8	0.4375839	0.209	0.989
128	8	0.3558822	0.374	0.809

# Workers	# Nodes	Sync/Comm ratio	Comm/Comp ratio
2	2	0.000211	0.07
4	2	0.0297973	0.106
8	2	0.1157873	0.457
16	4	0.09246	0.275
32	8	0.1832368	0.16
64	8	0.4375839	0.209
128	8	0.3558822	0.374

Figure 6 highlights a significant increase in the synchronization/communication ratio when employing more than 16 cores. Detailed analysis from the execution table reveals that, in these instances, the time allocated to synchronization surpasses that dedicated to communication. Consequently, to mitigate the impact of synchronization/communication overhead, it is recommended to limit the usage of cores to 16 when the matrix size is set at m=n=16000 with steps=32.

Furthermore, Figure 7 underscores the importance of adhering to this guideline to prevent synchronization/communication overhead. The observed ratio aligns closely with theoretical expectations, with the exception of scenarios involving 8 or 16 cores. Hence, to optimize performance and minimize overhead, it is advisable to refrain from utilizing 8 or 16 cores when the matrix size is m=n=16000 with steps=32. Beyond this threshold, the communication/computation ratio remains relatively acceptable.

In conclusion, based on the specifications and performance metrics, the optimal number of cores for efficiency appears to be 2 and 4.

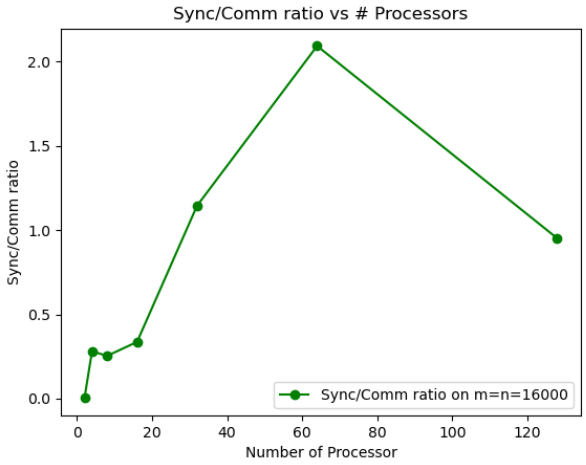

Fig. 6. Sync/Comm ratio vs Processors (m=n=16000, steps=32)

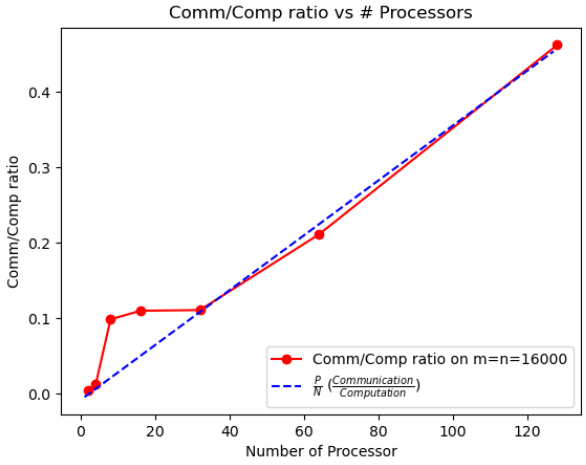

Fig. 7. Comm/Comp ratio vs Processors (m=n=16000, steps=32)

IX. Conclusion

In conclusion, the performance analysis of the parallel Game of Life (GOL) implementation using MPI on the DAS-5 cluster provides valuable insights. The implementation employs a 1D block partition method and faces challenges addressed through strategic adjustments.

The execution results demonstrate efficient parallelization, revealing speedup and efficiency improvements with an increasing number of workers. Notably, the order of border exchange and latency hiding techniques contribute to enhanced performance.

Advanced analyses showcase the impact of node configurations on efficiency, emphasizing the importance of distributing cores uniformly across nodes. Latency hiding further improves efficiency, particularly with larger grid sizes.

Overhead analysis highlights synchronization/communication and communication/computation ratios, guiding recommendations on the optimal number of cores for efficiency. Theoretical laws, Amdahl's and Gustafson's, provide context for assessing parallelization benefits.

In summary, the parallel GOL implementation demonstrates effective performance improvements, with considerations for node configurations and advanced techniques contributing to enhanced efficiency.