Performance Analysis of Parallel Game of Life Implementation in Julia

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

I. Problem Definition

Implement the parallelized version of the Game of Life in Julia. Then measure the speedup of it with benchmark code on the DAS-5 computing system. The analysis focuses on evaluating performance across different node and core configurations.

II. Implementation

A. create_chnls_snd_and_rcv

To establish communication channels, I employ a systematic indexing approach based on the relative positions within a 2x2 matrix. Specifically, the snd[i, j] channel of a worker is paired with the rcv[4-i, 4-j] channel on another worker, as illustrated in Figure 1.

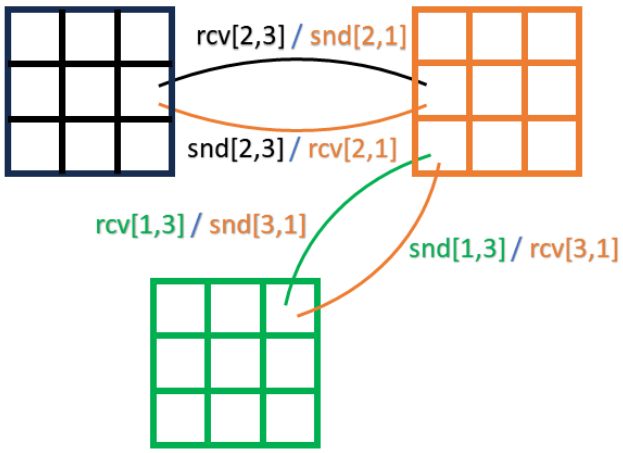

Fig. 1. Channel connections between workers

B. step_worker!

This function orchestrates a comprehensive update of GOL matrix. Initially, border exchange with neighboring workers is employed to update ghost cells. The communication process is streamlined by having each worker execute a put! operation in a specific direction, followed by a corresponding take! operation from the counterpoint, which optimizes the efficiency of the parallelized execution. The subsequent task involves updating GOL sub-matrices by the rules on each worker using the existing update! function.

C. game_worker

In this function, the primary objective is to ascertain the presence of a cycle within the submatrix updated by the step_worker! function and the two preceding historical submatrices. Upon detecting a cycle, the function transmits the value 1 to game_parallel_impl through the chnl_world channel. Conversely, in the absence of a cycle, the function transmits the value 0. Furthermore, the function terminates the loop and computes the wall time if it receives a stop signal from game_parallel_impl.

D. game_parallel_impl

This function is designed to aggregate has_cycled_par signals from chnl_world of all workers. Upon receiving a has_cycled_par signal with a value of 1 from each worker as the same iteration, it will initiate a coordinated termination of all workers (send chnl_stop with value 1 to all workers) and gather wall time information from each.

III. Challenges and Solutions

A. The Function of the Main Worker

Initially, I was influenced by the existing codebase, leading to a suboptimal approach where I gathered updated submatrices from workers and subsequently performed cycle detection, significantly impeding the program's performance. Upon reevaluation, I discerned that the desired outcome merely required verifying the correctness of the stopping iteration, unrelated to presenting the entire Game of Life (GOL) matrix.

Consequently, I refactored the implementation, empowering each worker to autonomously check for cycles and communicate only the final result to the main worker. This refinement yielded a substantial increase in program speedup.

B. Order in Border Exchange

In the initial phase, I initially aligned the put! and take! operations on each worker under the assumption that the ample channels would mitigate any potential blocking issues. However, upon scrutinizing the time allocated for border exchange on each worker, a noteworthy observation emerged. As illustrated in Figure 2, a consistent large difference in communication time between the fastest worker and itself was identified.

Upon further investigation, it became evident that certain workers experienced delays as they were compelled to wait

for the completion of all put! operations by others. This phenomenon was particularly pronounced for the snd[3,3]/rcv[1,1] channel, being the last to undergo put! yet the first to undergo take!. Subsequently, to address this inefficiency, I reconfigured the execution order of workers, ensuring that each worker performed a put! operation in a specific direction followed by a corresponding take! operation from the counterpart. This adjustment significantly optimized the efficiency of the parallelized execution.

```
From worker 3:    Communication time: 0.005000114440917969 seconds
From worker 4:    Communication time: 0.03600001335144043 seconds
From worker 5:    Communication time: 0.3470001220703125 seconds
From worker 2:    Communication time: 0.3470001220703125 seconds
```

Fig. 2. Large difference on ghost cell communication time between workers

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 game_worker function. In the execution of the parallel version (game_parallel_impl), the initial action is the invocation of @spawnat w game_worker. Consequently, the timer commences its tracking at the onset of the game_worker function. This timing encapsulates the entire duration of game_worker, including communication time throughout all iterations. Additionally, the final take! operation also occurs within the game_worker. 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 slowest worker in each run is identified, and the minimum across all runs is taken 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 70000 are impractical. For the larger size, I have opted for 64000, a multiple of 128 and taking around 70 sec. Consequently, the medium-sized configuration is set at half of this value, i.e., 32000.

E. Fixed size with m=n=32000

1) Serial version result:

Run 1	12.21397
Run 2	12.20083
Run 3	12.21374
Run 4	12.20770
Average: 12.20906	

2) Parallel version results:

# Workers	# Nodes	Exec time	Efficiency	Speedup
2	2	6.60587	0.92411	1.84821
4	2	7.48027	0.40804	1.63216
8	2	5.11752	0.29822	2.38573
16	4	3.10884	0.24545	3.92720
32	8	1.62628	0.2346	7.50732
64	8	1.11469	0.17114	10.95285
128	8	1.02639	0.09293	11.89507

F. Fixed size with m=n=64000

1) Serial version result:

Run 1	80.08023
Run 2	69.91987
Run 3	65.18750
Run 4	73.50810
Average: 72.17392	

2) Parallel version results:

# Workers	# Nodes	Exec time	Efficiency	Speedup
2	2	26.32445	1.37085	2.74170
4	2	24.33985	0.74131	2.96525
8	2	17.13667	0.52646	4.21166
16	4	9.99976	0.4511	7.21756
32	8	6.55125	0.34428	11.01681
64	8	4.04775	0.2786	17.83061
128	8	3.49930	0.16113	20.62520

V. Performance Analysis

A. Communication/computation ratio

To assess performance, I commence by quantifying the communication and computation aspects of the program. (Let us set N = max(m,n) here.) Employing a 2D block partition method, we update N^2/P items per iteration. In this context, communication involves obtaining data from eight neighbors, translating to two messages of size $N/\sqrt{P}$ and $N/\sqrt{P}$, as well as four messages of one item each per iteration. Consequently, the communication/computation ratio asymptotically approaches $O(\sqrt{P}/N)$.

The ratio of $O(\sqrt{P}/N)$ means, in this parallel framework, the addition of processors is anticipated to yield diminishing marginal utility. To validate this theoretical premise against practical experience, I chart the derivative of the communication/computation ratio alongside the efficiency. Figure 3 illustrates a consistent trend, affirming the alignment with my theoretical assumption.

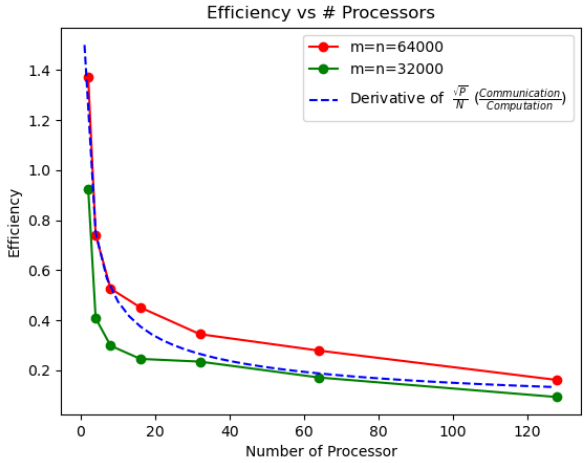

Fig. 3. Efficiency on sizes: m=n= 32000 & 64000

B. Speed up with both sizes

The speedup analysis, depicted in Figure 4 for varying numbers of workers, reveals an intriguing observation. Obviously, there are two flattening of the speed-up curve, which follows what we have discussed on communication/computation ratio. Specifically, as assumed, the size 32000 exhibits diminishing marginal utility earlier, given its relatively smaller scale.

Interestingly, Figure 4 illustrates that employing 2 workers outperforms the performance achieved with 4 workers, particularly when applied to the smaller size. A closer examination of the program suggests that with only two workers, each worker connects two sides of communication directly to itself, minimizing idle time and allowing for enhanced scalability benefits. This phenomenon is supported by the observed number of communications ($O(n/\sqrt{P})$), indicating the occurrence of this unique case, notably under a smaller size when P is equal to 2. That's why it happens on size 32000 intsead of size 64000.

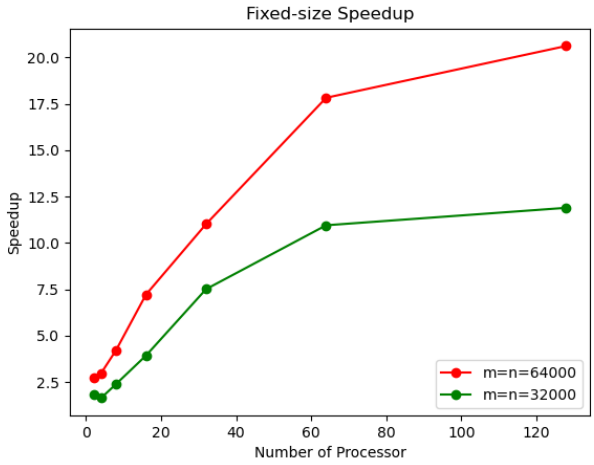

Fig. 4. Speed up on sizes: m=n= 32000 & 64000

C. Efficiency analysis

Remarkably, a superlinear speedup is observed in Figure 3 when utilizing 2 workers with a size of 64000. Consistent with theoretical assumptions, this phenomenon significantly benefits from scalability, given the $O(\sqrt{P}/n)$ communication/computation ratio and the presence of a large size with a small number of processors. Furthermore, as previously discussed, the direct connection of two sides of communication to each worker itself minimizes idle time, providing a sensible explanation for the observed superlinear behavior under these conditions.

VI. Advanced analysis

A. Same # cores on 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=24000		
# Workers per node	# Nodes	Exec time
16	1	2.8148
8	2	1.8711
4	4	1.7317
2	8	1.6210
1	16	1.4867

m=n=50000		
# Workers per node	# Nodes	Exec time
16	1	9.5625
8	2	7.4110
4	4	6.1863
2	8	5.7465
1	16	5.9271

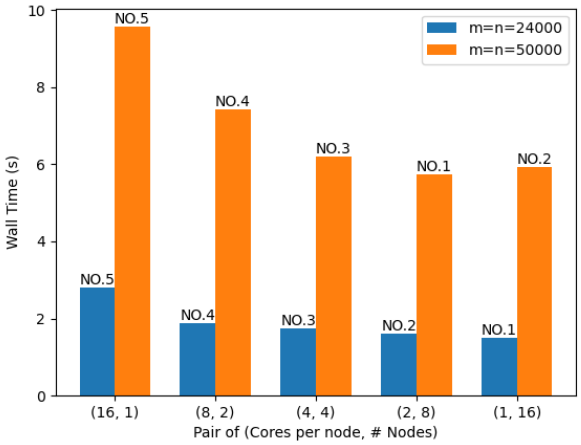

Fig. 5. Same # cores on different # nodes

C. Analysis

In Figure 5, it is evident that the optimal configuration consistently involves utilizing a greater number of nodes with fewer cores per node. Conversely, the least favorable configuration is observed when concentrating all cores within a single node.

This performance discrepancy arises due to the mode of data transfer within the same node, which leverages shared memory. Consequently, aggregating all cores in a single node results in a considerable execution time. In contrast, inter-node communication employs the "FDR" InfiniBand, a high-speed interconnect network. This distinction underscores the efficiency gained by minimizing the number of cores per node, resulting in improved overall execution times.

VII. Communication Overhead Analysis

A. Communication time

In evaluating communication time, the primary metric involves measuring the time allocated to update_ghost_parallel!. Additionally, the exchange of stop signal between root and workers is timed within each iteration to capture its contribution.

B. Computation time

The computed computation time is derived by subtracting the accumulated communication time from the overall timing results.

C. Execution results

# Workers	# Nodes	Comm time	Comp time
2	2	0.653	5.867
4	2	3.672	3.711
8	2	4.001	1.05
16	4	2.350	0.718
32	8	1.140	0.465
64	8	0.744	0.356
128	8	0.718	0.295

# Workers	# Nodes	Comm/Comp ratio
2	2	0.111
4	2	0.989
8	2	3.811
16	4	3.271
32	8	2.451
64	8	2.089
128	8	2.434

D. Analysis

In Figure 6, the observed ratio aligns with theoretical expectations, with the exception of scenario involving 8, 16, or 32 cores. Hence, to optimize performance and minimize overhead, it is advisable to refrain from utilizing these number of cores when the matrix size is m=n=32000. Beyond this threshold, the communication/computation ratio remains relatively acceptable.

In conclusion, based on this specifications, the discussion in other sectios and performance metrics, the optimal and most economical number of cores for efficiency appears to be 2 and 4.

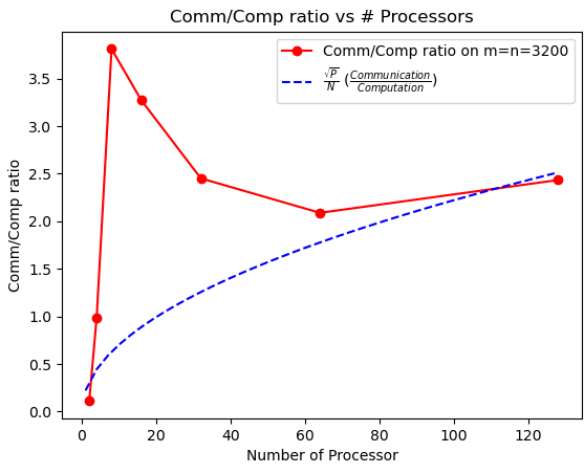

Fig. 6. Comm/Comp ratio vs Processors (m=n=32000)

VIII. Julia vs C+MPI performance analysis

A. Comparison of execution time (m=n=16000)

# Workers	# Nodes	Juila	MPI
2	2	2.038	17.0265
4	2	2.311	8.7165
8	2	1.599	4.88875
16	4	0.848	2.69
32	8	0.457	1.78575
64	8	0.580	1.5415
128	8	0.764	1.86625

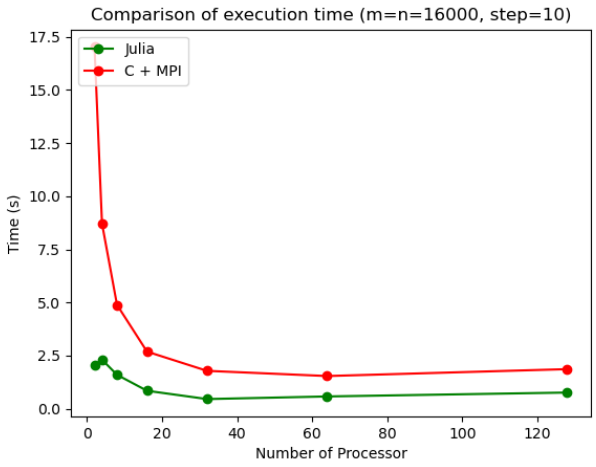

Fig. 7. Comparison of execution time (m=n=16000, step=10)

B. Comparison of execution time (m=n=32000)

# Workers	# Nodes	Juila	MPI
2	2	6.60587	22
4	2	7.48027	12.1705
8	2	5.11752	6.9345
16	4	3.10884	4.2865
32	8	1.62628	3.1935
64	8	1.11469	1.859
128	8	1.02639	197.176

C. Analysis

After conducting a performance analysis between Julia and C+MPI, the results consistently demonstrate that Julia outperforms C+MPI. This discrepancy may be attributed to the distinct algorithms employed in each program. Notably, Julia utilizes a 2D partition approach, whereas C+MPI employs a 1D partition strategy.

The impact of this algorithmic disparity becomes particularly pronounced when dealing with relatively small dataset sizes. Moreover, as the dataset size scales up, the C+MPI system experiences an inexplicable slowdown. In light of these findings, it is evident that opting for the Julia implementation consistently proves to be the superior choice. Therefore, the recommendation is to consistently choose the Julia implementation, considering its superior performance across varying dataset sizes.

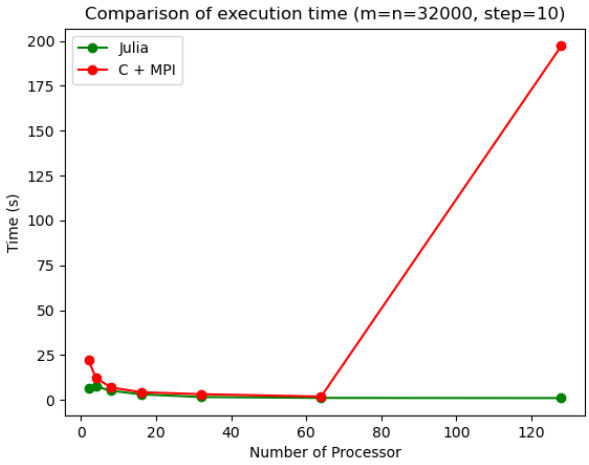

Fig. 8. Comparison of execution time (m=n=32000, step=10)

IX. Conclution

In conclusion, our study successfully implemented and analyzed a parallelized version of the Game of Life in Julia, focusing on performance across different node and core configurations. We addressed challenges, such as suboptimal approaches, and optimized the execution order to enhance overall performance. The analysis revealed intriguing observations, including diminishing marginal utility under specific conditions and superlinear speedup with two workers for larger sizes. A comparative analysis with C+MPI demonstrated Julia's consistent superiority. Our findings recommend the use of Julia, particularly for smaller dataset sizes, and highlight the impact of node configurations on efficiency. While limitations exist, this work contributes valuable insights into parallel GOL implementations and underscores Julia's effectiveness in achieving efficient parallelization.