IMPLEMENTATION OF OPEN-MP VERSION OF THE GAME OF LIFE

Homework 3

By
Ali Takallou

HIGH PERFORMANCE COMPUTING
CS 581
OCTOBER 12, 2024
THE UNIVERSITY OF ALABAMA

1 Problem Specification

Many fields, including biology, physics, philosophy, and computer science, can use Conway's Game of Life, a cellular automaton simulation algorithm, to explore evolutionary concepts. The mathematician John Conway developed it in 1970 to explore self-reproducible systems proposed by John von Neumann, the namesake of today's dominant computer architecture. The game follows a set of basic rules that control the evolution of living cells on the game board. The next generation updates the game board based on the current generation's state of the cell and its neighbors. If a cell is alive in the current generation, it survives to the next generation if it has two or three neighbors; otherwise, it dies of loneliness or overcrowding. If a cell is dead in the current generation, it can spontaneously generate a living cell in the next generation if it has exactly three neighbors in the current generation. These simple rules allow us to explore a rich variety of physical systems through the game of life. Game boards can generate patterns that behave in ways that suggest intelligent visual patterns. This provides an opportunity to delve into the essence of patterns, their interpretation, and the possibility that biological patterns are merely manifestations of basic, fundamental principles, with the observer imbuing them with meaning rather than the pattern itself. In this assignment, we aim to implement a parallelized version of the Game of Life. This framework allows for customization of board size, thread count, and simulation runtime, enabling an in-depth exploration of the performance characteristics of parallelized programs. The solution presented in this report accepts five arguments: board size, maximum number of iterations, dimensions of the parallel region (rows and columns), and an output path for saving the final board state. The simulation concludes either when the game board stabilizes across generations or reaches the specified maximum iteration. These rules are applied consistently across the matrix as the game progresses through each generation.

In Homework 1, we conducted various tests on the serial version of the Game of Life program using our personal lab computer. The primary objective of this assignment is to design and test a multithreaded version of the Game of Life using OpenMP. This involves verifying the functionality and correctness of the program across different thread counts. Therefore, given the same initial conditions, the final board state of the multithreaded version should match that of the single-threaded serial program. Additionally, we will use both the Intel and GCC compilers to provide a comprehensive comparison of the multithreaded program's performance.

2 Program Design

To manage the multithreaded implementation of the program and prevent race conditions, we create a parallel region where each thread is responsible for a specific portion of the Game of Life board. Each thread computes the flags for its assigned region independently. Figure 1 illustrates how this regional parallelization is implemented, demonstrating the progression of the Game of Life through successive generations.

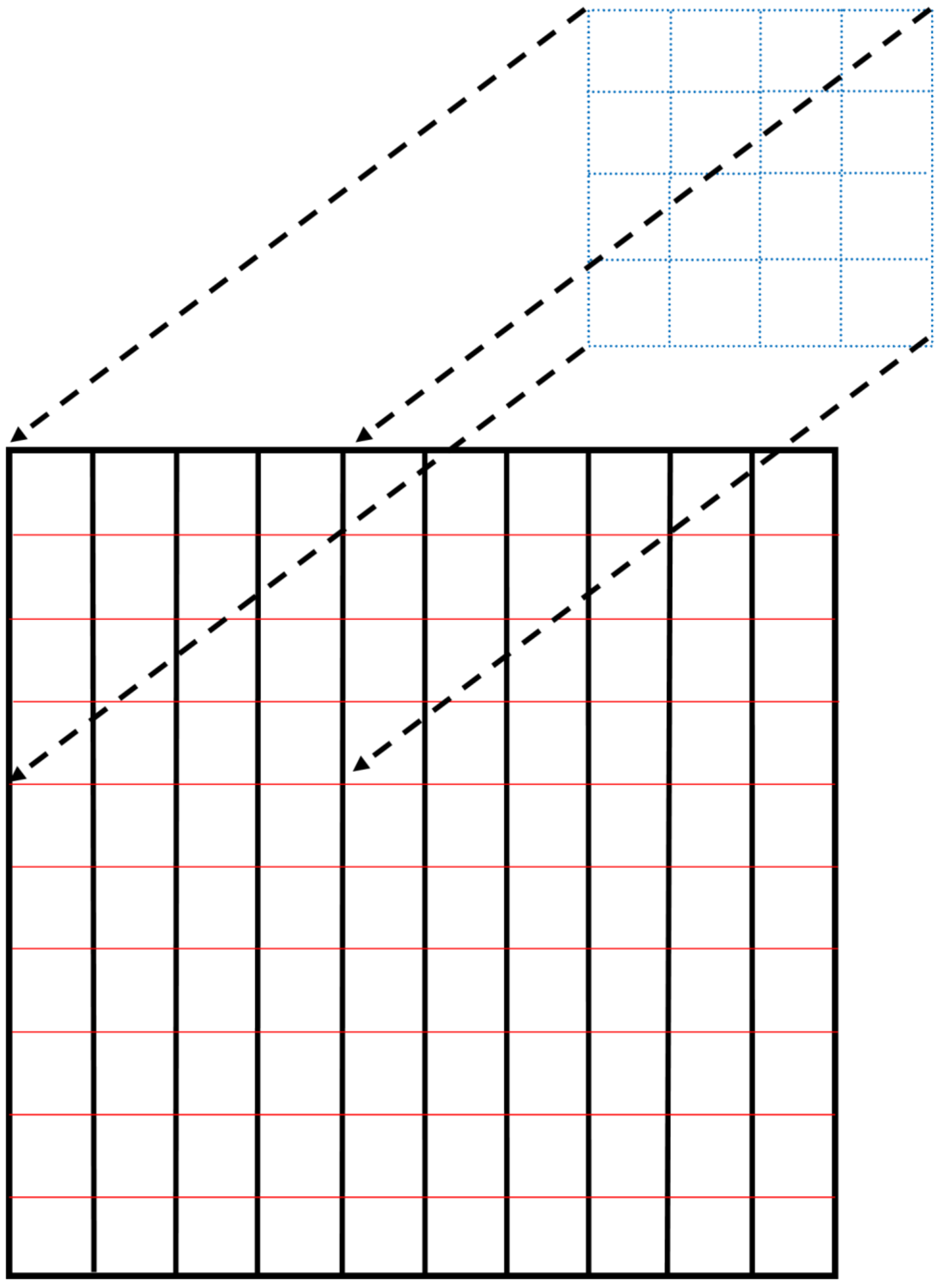
Figure 1 - Regional parallelization of Game of Life

The program is designed so that each thread computes a specific region of the next-generation board in the Game of Life. A critical aspect of OpenMP programming is properly defining private and shared variables for the threads. In this implementation, shared variables include the current generation matrix, the next generation matrix, the matrix size, the number of rows and columns in the parallel region, and the count of cells that need to be updated. Each thread also has private variables: the addresses of the cells for which they are calculating neighboring values and the count of alive cells surrounding each cell.

3 Testing Plan

In Homework 2, we evaluated the performance of the Game of Life on the ASAX cluster. For this assignment, we plan to compile and execute the multithreaded version of the Game of Life on the ASAX cluster. The system specifications for a core in the ASAX cluster are as follows:

Machine name:	asaxg005.asc.edu
OS Name:	Linux
OS Version:	4.18.0-513.9.1.el8_9.x86_64
processor name:	AMD EPYC 7713 64-Core Processor
Maximum Clock Speed:	1931.266
Total Physical Memory:	1.0Ti
Compiler name and version:	icx Intel(R) oneAPI DPC++/C++ Compiler 2023.1.0
	gcc (GCC) 8.5.0 20210514 (Red Hat 8.5.0-20)
Compiler Flags:	-o and -O3

To evaluate the performance improvement of the OpenMP program, we plan to conduct multiple tests using a constant problem size while varying the number of threads. Specifically, we will run the program with a board size of 5000 and 5000 iterations, testing with 1, 2, 4, 8, 10, 16, and 20 threads. Each configuration will be executed three times, and we will report the average execution time. To further assess performance improvement, we will calculate the speedup and efficiency indices. Additionally, we will compile the program with both the Intel and GCC compilers to evaluate the OpenMP program's performance across different compilers. To establish a regional thread based on the thread count, it is essential to define the rows and columns within that region. The following table outlines the number of rows and columns for each test case.

Table 1- Number of Rows and Columns based on the number of threads

Number of threads	Number of Rows	Number of Columns
1	1	1
2	2	1
4	2	2
8	4	2
16	8	2
20	10	2

To gain a comprehensive understanding of the program's runtime improvement, we are calculating the speedup and efficiency indices. The **speedup index** measures the improvement in runtime achieved by a parallel program compared to its serial (single-processor) version. It is calculated by dividing the serial runtime by the parallel runtime with multiple processors, using the formula $S = \frac{\mathrm{Ts}}{\mathrm{Tp}}$where Ts is the serial runtime, and Tp is the parallel runtime. A higher speedup indicates better performance, with the ideal case being that the speedup matches the number of processors used. The **efficiency index** evaluates how effectively the processors are utilized in a parallel program. It is derived from the speedup by dividing it by the number of processors, represented as E=S/P where P is the number of processors. Efficiency values range from 0 to 1, with values closer to 1 indicating that the processors are contributing effectively, maximizing resource utilization and minimizing overhead.

4 Test Cases

Table 2- Performance Results on asax cluster using intel compiler

Test Case	Problem size	Number of threads	Experiment 1	Experiment 2	Experiment 3	Average time
1	5000 * 5000	1	295.15	296.98	298.46	296.8633
2	5000 * 5000	2	155.97	181.41	178.41	171.93
3	5000 * 5000	4	128.20	137.91	160.21	142.1067
4	5000 * 5000	8	73.26	101.70	85.86	86.94
5	5000 * 5000	10	70.15	100.87	86.65	85.89
6	5000 * 5000	16	64.567025	73.4447	79.288835	72.43352
7	5000 * 5000	20	86.009559	78.399505	76.639089	80.34938433

Table 3- Performance Results on asax cluster using gcc compiler

Test Case	Problem size	Number of threads	Experiment 1	Experiment 2	Experiment 3	Average time
1	5000 * 5000	1	323.11	324.08	324.19	323.793
2	5000 * 5000	2	186.72	186.58	185.76	186.353
3	5000 * 5000	4	106.80	103.05	113.52	107.79
4	5000 * 5000	8	75.672	43.009	59.472	59.384
5	5000 * 5000	10	50.168	53.553	60.782	54.834
6	5000 * 5000	16	33.818171	31.493692	31.563225	32.292
7	5000 * 5000	20	44.811900	57.297876	85.376275	62.495

The performance results from two compilers, Intel and GCC, on the Asax cluster were recorded for a problem size of 5000×5000, using varying numbers of threads. The primary objective was to observe the impact of thread count on execution time and to identify any performance variations between the Intel and GCC compilers under similar conditions.

In Table 2, which shows the performance results using the Intel compiler, we observe that the average execution time for a single thread was approximately 296.86 seconds. This single-threaded result provides a baseline for comparing the performance gains achieved with additional threads. As the number of threads increased from 1 to 16, there was a consistent reduction in the average execution time, indicating that the Intel compiler handles parallel processing effectively. For example, with 2 threads, the average time dropped to 171.93 seconds, representing a noticeable improvement. At 4 threads, the average time further decreased to 142.11 seconds, while 8 threads saw the execution time fall to 86.94 seconds. The improvement continued with 10 threads, achieving an average of 85.89 seconds, although the marginal gains between 8 and 10 threads suggested a point of diminishing returns. When the thread count reached 16, the average time reduced further to 72.43 seconds, showing that the Intel compiler benefits significantly from higher parallelism up to this point. However, with 20 threads, the average time unexpectedly increased to 80.35 seconds, which implies that using more threads than necessary may introduce additional overhead or inefficiencies. From these observations, it appears that the Intel compiler performs optimally with thread counts between 8 and 16 for this problem size on the Asax cluster.

In Table 3, the performance results using the GCC compiler reveal a different pattern. The single-threaded execution time with GCC was 323.79 seconds, which is noticeably slower than the Intel compiler's single-thread performance, suggesting that the Intel compiler might be better optimized for single-threaded tasks on this hardware. However, as the thread count increased, the GCC compiler showed more dramatic improvements. For instance, using 2 threads, the average execution time dropped to 186.35 seconds, although this was still slightly higher than the Intel compiler's performance with the same number of threads. At 4 threads, the GCC compiler reduced the time further to 107.79 seconds, demonstrating effective scaling with parallelism. When 8 threads were used, the average time fell to 59.38 seconds, which was lower than the Intel compiler's performance at the same level. With 10 threads, the GCC compiler achieved an average of 54.83 seconds, indicating continued improvement without the diminishing returns seen with the Intel compiler. The GCC compiler reached its best performance at 16 threads, with a low average time of 32.29 seconds, demonstrating its superior scalability with higher thread counts. At 20 threads, however, the execution time increased slightly to 62.50 seconds, suggesting that the GCC compiler, like the Intel compiler, might encounter overhead or inefficiencies when using more threads than optimal for this specific task.

Comparing the two compilers, the Intel compiler had a faster single-threaded execution time than GCC, indicating it may be better optimized for single-threaded tasks. In multi-threaded scenarios, however, the GCC compiler demonstrated better scalability, achieving lower execution times at higher thread counts, particularly between 8 and 16 threads. For both compilers, 16 threads appeared to yield the best execution times, although at 20 threads, both showed an increase in time, indicating potential overhead or bottlenecks, possibly related to the hardware configuration or the way each compiler handles thread management. Overall, while the Intel compiler performed better in single-threaded execution, the GCC compiler showed superior performance in multi-threaded settings, particularly as the number of threads increased. These results suggest that the choice of compiler can significantly impact performance when leveraging multi-threading capabilities on the Asax cluster, and they highlight the importance of selecting an appropriate thread count to optimize execution time for the specific compiler and hardware configuration.

Figure 2- Multithread Game of Life Performance

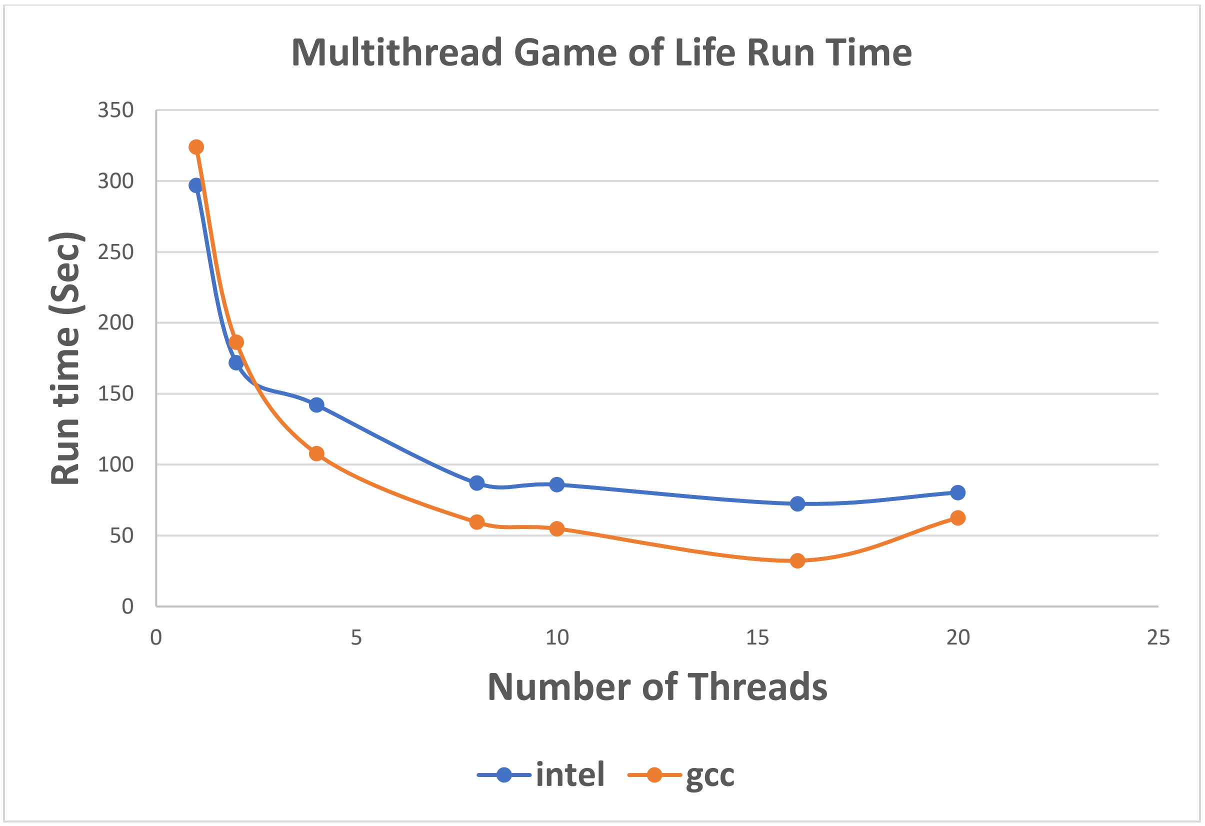

Figure 2 shows the performance improvement of multithread program of Game of life by adding more thread to execute the program. It can be observed that the number of threads and the runtime have nonlinear relationship together.

Table 4- efficiency and Speed-up table based on the number of threads

	speed up	efficiency
1	1	1
2	0.579155	0.289578
4	0.478694	0.119674
8	0.292862	0.036608
10	0.289325	0.028933
16	0.243996	0.01525
20	0.270661	0.013533

Figure 3- Speed-up and efficiency

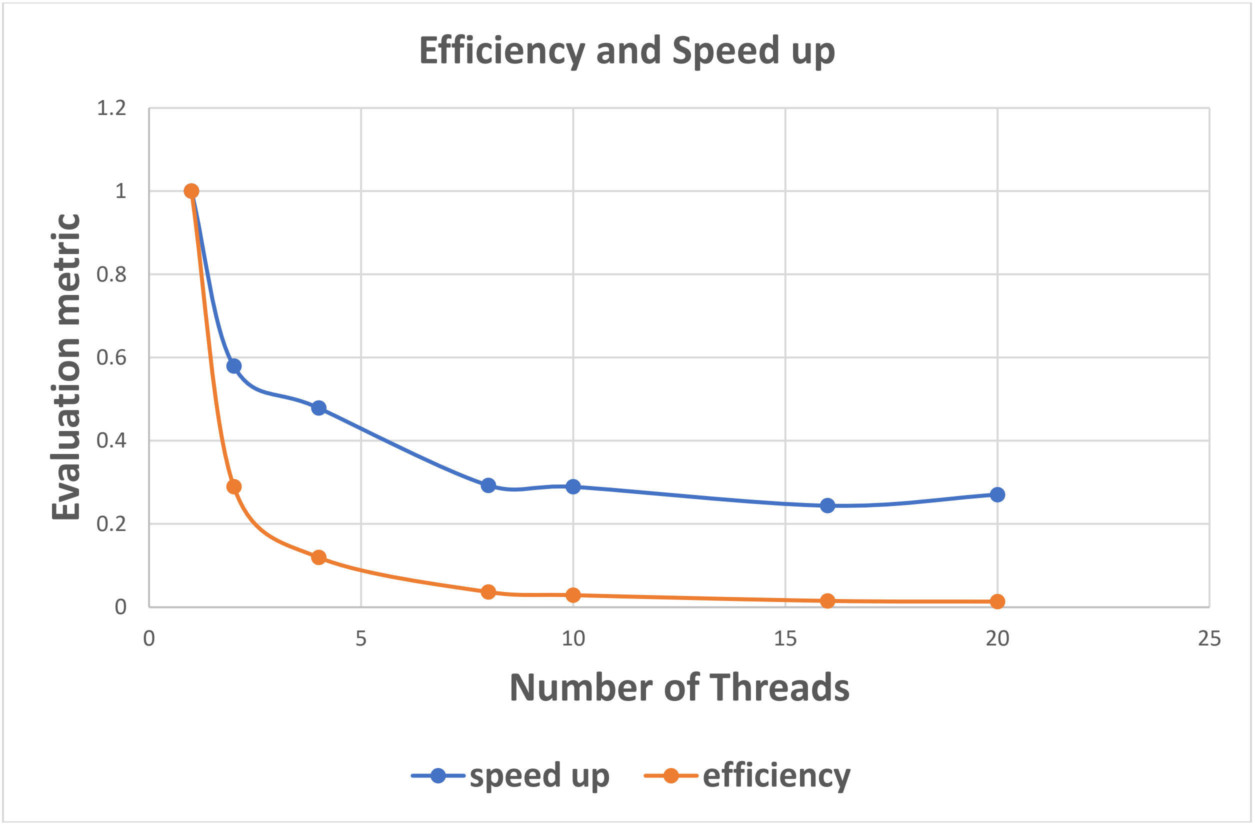

Table 4 presents the values of speedup and efficiency as the number of threads increases from 1 to 20. The data indicates that both speedup and efficiency decline as the number of threads rises, suggesting diminishing returns on runtime improvement with the addition of more threads.

The speedup starts at 1 with a single thread, establishing the baseline. When using 2 threads, the speedup is approximately 0.58, which is notably below the ideal of 2. This downward trend continues as the thread count increases, with speedup values dropping below 0.3 by the time 8 threads are used. These non-ideal speedup values indicate that adding more threads does not proportionally enhance runtime, likely due to overhead factors such as increased communication, synchronization delays, or inefficiencies in workload distribution.

Efficiency, on the other hand, also begins at 1 with a single thread, reflecting full resource utilization. However, by the time 2 threads are used, efficiency drops to about 0.29, showing that each additional thread contributes less effectively. Efficiency continues to decrease steeply as the thread count grows, reaching as low as 0.01 by 20 threads. This sharp decline suggests that the added threads yield diminishing returns, with significant overheads preventing efficient utilization of computational resources.

Figure 3 graphically illustrates these trends observed in Table 4. The speedup curve (in blue) and the efficiency curve (in orange) both exhibit steep declines as the number of threads increases. The curves flatten at higher thread counts, with efficiency remaining extremely low and speedup offering only marginal improvements or even showing slight decreases beyond a certain point.

In interpreting these results, it is evident that the parallelization reaches a point where additional threads contribute little to no advantage due to the overhead associated with parallel execution. The low efficiency at higher thread counts highlights that resources are underutilized, potentially due to excessive synchronization, communication costs, or an imbalance in workload distribution. From these trends, it becomes clear that an optimal number of threads likely lies below the maximum tested here. Using fewer threads would potentially yield a more balanced improvement in speedup while avoiding the inefficiencies associated with excessive parallelism. Therefore, Table 4 and Figure 3 collectively show that while some speedup is achieved through parallelization, both speedup and efficiency exhibit rapid declines as thread counts increase, underscoring the importance of finding an optimal level of parallelism for the workload in question.

5 Analysis and Conclusion

The results demonstrate the runtime improvement of the Game of Life using Open MPI and a parallel design (Gardner, 1970; Johnston & Greene, 2022). The program achieves parallelization by defining specific rows and columns to establish regions, allowing each thread to compute the next generation independently. Open MPI automates the parallelization of loops, distributing threads across these regions efficiently. This report also compares the performance of GCC and Intel compilers in the multithreaded version of the Game of Life. While the Intel compiler outperformed GCC in single-threaded execution, GCC showed slightly better overall performance in multithreaded scenarios. A key aspect of implementing the Game of Life with Open MP involves the careful design of shared and private variables. Shared variables can be accessed and modified by all threads simultaneously. In this program, the number of changed cells serves as a shared variable, where each thread adds to its value when a cell in its region changes. While adding threads can reduce the simulation runtime, the relationship is nonlinear, making it challenging to predict runtime for a multithreaded program. We also observed that, in some cases, increasing the thread count beyond 16 to 20 resulted in longer runtimes, indicating unexpected overhead. Additionally, all test cases were executed on the Asax cluster. Due to resource availability, we parallelized our test cases across both cores and jobs.

Our github reposity for this project is located at
"https://github.com/atakallou/Fall2024_CS581_HW3"

6 References

Denning, P. J., & Lewis, T. G. (2016). Exponential laws of computing growth. *Communications of the ACM*, *60*(1), 54–65.

Gardner, M. (1970). Mathematical games. *Scientific American*, *222*(6), 132–140.

Johnston, N., & Greene, D. (2022). *Conway's Game of Life: Mathematics and Construction*. Nathaniel Johnston.