Game of Life 3D Project Report Group 20

Miguel Alves, Pedro Rodrigues, Renato Martins

1 INTRODUCTION

The Game of Life has captivated generations with its simple rules and complex emergent behavior. In its original two-dimensional form, the game simulates the evolution of cells on a grid, governed by rules of birth, death, and survival based on the number of neighboring cells.

Expanding upon this classic concept, our project ventures into the realm of three-dimensional Game of Life simulations. In this 3D variant, cells exist within a volumetric space, adding another dimension of complexity to the simulation. With each cell's fate influenced by its 26 neighboring cells, the dynamics of the 3D Game of Life offer intriguing possibilities for exploration and analysis.

In this report, we detail our efforts to parallelize the 3D Game of Life simulation using OpenMP, a widely-used API for shared-memory parallel programming.

2 OPTIMIZATIONS ON SERIAL VERSION

In optimizing the code structure for parallelization, strategic changes were made to move the parallelization focus from the `evolve_cell` function to the higher-level `simulation` function in the transition from the serial to OpenMP version offers several benefits. By parallelizing at the simulation level, we achieve coarser-grained parallelism, reducing synchronization overhead and improving load balancing across threads. This approach simplifies the code structure, encapsulating the entire parallelization logic within a single function and facilitating code maintenance. Additionally, parallelizing the simulation process enables efficient task-level parallelism, where each thread independently executes the grid evolution process, leading to improved scalability and performance.

3 OPENMP APPROACH

Our parallelization approach involves optimizing key components of the simulation, such as cell evolution, to leverage the capabilities of parallel computing effectively. We address synchronization concerns to ensure data integrity and explore load balancing strategies to maximize performance across multiple threads.

4 DECOMPOSITION

The code uses a spatial decomposition approach, dividing the three-dimensional grid into smaller chunks handled by individual threads. This decomposition strategy is evident in the parallelization of the `simulation` function, where the `evolve_cell` function is applied to each cell of the grid in parallel. The grid is partitioned into smaller blocks, with each thread responsible for processing a subset of these blocks. This spatial decomposition ensures that the workload is evenly distributed

- *Miguel Alves, nr. 95650,*
  E-mail: miguel.paraiso.alves@tecnico.ulisboa.pt,
- *Pedro Rodrigues, nr. 99300,*
  E-mail: pedro.dias.rodrigues@tecnico.ulisboa.pt,
- *Renato Martins, nr. 102314,*
  E-mail: student3@tecnico.ulisboa.pt,
  Instituto Superior Técnico, Universidade de Lisboa.

among threads, optimizing parallel execution and load balancing. Additionally, the use of private arrays for each thread to store species counts and the subsequent reduction operation to compute global counts further enhances the decomposition strategy, enabling efficient parallel processing while maintaining data consistency. Overall, this spatial decomposition approach maximizes parallelism and minimizes synchronization overhead, contributing to improved performance in the Game of Life 3D simulation.

5 Synchronization Concerns

The synchronization concerns in the provided code primarily revolve around ensuring data consistency and preventing race conditions when multiple threads access shared resources.

1) Critical Sections: The code employs critical sections using `#pragma omp critical` to ensure that certain blocks of code are executed atomically, preventing multiple threads from accessing shared variables simultaneously. For example, in the `simulation` function, where species counts are updated, a critical section is used to ensure that the `global_count_per_generation` array is updated safely.
2) Global Reduction: To calculate global counts of species populations across all threads, a reduction operation is performed after each generation. This operation accumulates counts from private arrays maintained by each thread into a global array, ensuring that the final counts are accurate.
3) Memory Access: As threads read from and write to shared memory locations, proper synchronization is crucial to avoid data races. For instance, in the `gen_initial_grid` function, where cells of the grid are initialized based on random density, atomic operations are used to update species counts to prevent race conditions.

Overall, these synchronization mechanisms ensure that concurrent operations on shared resources are properly coordinated, preserving data integrity and preventing inconsistencies that could arise in a parallel execution environment.

6 Load Balancing

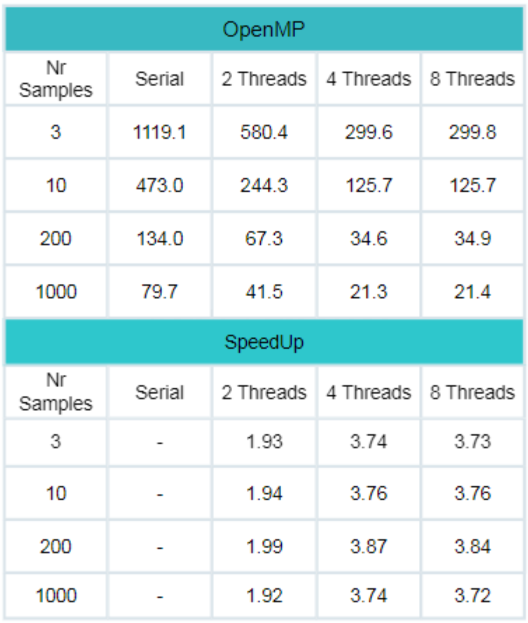
Load balancing in the provided code is addressed through the even distribution of workload among threads during parallel execution. This ensures that each thread has a roughly equal amount of work to perform, maximizing parallel efficiency and minimizing idle time.

1) Loop Parallelization: The code parallelizes loops using OpenMP directives, such as `#pragma omp for`, to distribute iterations among threads. For example, in the `simulation` function, the main loop iterating over cells of the grid is parallelized, allowing multiple threads to process different regions of the grid concurrently.
2) Work Distribution: Within parallel regions, work is distributed evenly among threads using default scheduling. For instance, in the `simulation` function, the nested loop iterating over grid cells is parallelized with the `collapse(3)` clause, ensuring that each thread processes a contiguous block of grid cells.
3) Private Data: Certain data structures, such as private arrays for species counts per thread, are used to maintain thread-local data. This prevents contention for shared resources and ensures that each thread can independently update its own data without synchronization overhead.

By distributing work evenly among threads and minimizing overhead associated with synchronization and data sharing, the code achieves effective load balancing, enabling efficient parallel execution and maximizing utilization of available computational resources.

7 Performance Results

After running the code with 1 (Serial), 2, 4 and 8 available threads, we obtained the following results from a computer in the L01 laboratory of the Pavilhão de Informática (https://rnl.tecnico.ulisboa.pt/laboratorios/lab-1/). These computers have a Intel(R) Core(TM) i5-7500 CPU with a 4-core processor.



OpenMP				
Nr Samples	Serial	2 Threads	4 Threads	8 Threads
3	1119.1	580.4	299.6	299.8
10	473.0	244.3	125.7	125.7
200	134.0	67.3	34.6	34.9
1000	79.7	41.5	21.3	21.4

SpeedUp				
Nr Samples	Serial	2 Threads	4 Threads	8 Threads
3	-	1.93	3.74	3.73
10	-	1.94	3.76	3.76
200	-	1.99	3.87	3.84
1000	-	1.92	3.74	3.72

Figure 1. Time Execution and SpeedUp Calculation

Based on the provided performance results, it's evident that the parallelization strategy effectively improves the performance of the Game of Life simulation. Here are some points based on the information provided:

- Optimal Thread Count: The performance results indicate that the optimal number of threads for achieving the best speedup is 4. This suggests that the workload is efficiently distributed among the available threads, and the overhead of coordinating additional threads outweighs the benefits gained from parallelization beyond 4 threads. The outcomes obtained conform to expectations, as evidenced by the observed enhancement from utilizing 2 threads to 4, followed by a subsequent stagnation upon employing 8 threads. This phenomenon is explained by the limitation of the CPU's core count to 4, necessitating the allocation of each thread to a distinct core. Given the absence of 8 cores and despite the presence of Hyper-threading functionality in the CPU, the performance fails to exhibit improvement.

- Speedup Analysis: Achieving a speedup of approximately 3.8 with 4 threads showcases a significant improvement compared to the serial implementation. This result demonstrates the efficiency of parallelization, leveraging multiple cores to execute the simulation much faster. When testing with 2 threads, we observed speedups slightly below 2, which represents the best achievable value in this scenario.

- Performance: The speedup for all examples suggests that the parallelized implementation performs well with the problem size and number of threads. It shows that increasing the problem size (e.g., larger grid dimensions and more generations) while maintaining the same number of threads can still achieve great performances.

8 Conclusion

In conclusion, the parallelization of the 3D cellular automaton simulation using OpenMP frameworks has been successful in improving performance and scalability. The approach of utilizing shared-memory parallelization techniques has enabled efficient utilization of computational resources across different architectures. However, further optimization and tuning may be required to address performance bottlenecks and achieve optimal parallel efficiency.