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 and MPI frameworks.

2 OpenMP + MPI Approach

The provided code implements a parallelized version of the 3D Game of Life simulation using both OpenMP and MPI for parallelization. While creation of this new version, no optimizations were made. In the OpenMP approach, parallelism is introduced at the thread level within a shared-memory environment. This approach allows multiple threads to collaborate on the simulation tasks, thereby leveraging the computational resources available within a single node efficiently. The simulation function is parallelized using OpenMP directives, enabling concurrent execution of the simulation logic across multiple threads. Loop-level parallelism is achieved to distribute the workload among threads effectively, with loops iterating over the grid cells parallelized using OpenMP directives such as `#pragma omp for`.

To ensure data integrity and prevent race conditions when multiple threads access shared resources, synchronization mechanisms such as critical sections `#pragma omp critical` are employed. Critical sections are utilized to update global data structures, such as arrays tracking species counts and maximum generations, safely and atomically. Additionally, atomic operations `#pragma omp atomic` are used to perform atomic updates on shared variables, minimizing contention among threads. Private data structures, such as arrays for species counts per thread, are utilized to maintain thread-local data without contention for shared resources.

In contrast, the MPI approach introduces process-level parallelism in a distributed-memory environment. Each MPI process operates independently and communicates with other processes through message passing. Each process is responsible for a portion of the grid that it generates and saves locally, but all the processes run the random seed generator the

- *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: renato.martins@tecnico.ulisboa.pt,
  Instituto Superior Técnico, Universidade de Lisboa.

same amount of times and by the same order, to ensure that the grids are always the same. Data distribution is facilitated using MPI communication primitives such as `MPI_Isend` or `MPI_Recv`, allowing each process to generate its own portion of the grid and only send the required parts of the grid to other processes. All the process require the borders of the rank-adjacent processes to ensure that they hold information about all the neighbours of all the cells of the grid that they are responsible for.

After each generation, global reduction operations are performed using MPI reduction operations `MPI_Reduce` to gather information from all processes efficiently. This includes computing the sum of the occurrences for all the species for that generation, so that the global statistics such as maximum species counts and corresponding generations can be calculated at the end. By combining OpenMP for shared-memory parallelism and MPI for distributed-memory parallelism, the code achieves enhanced performance and scalability, making it suitable for execution on multi-core CPUs and distributed computing clusters.

3 Decomposition

In the MPI approach employed in the provided code for the parallelized 3D Game of Life simulation, decomposition plays a crucial role in distributing the computational workload among MPI processes efficiently. Decomposition involves partitioning the three-dimensional grid representing the simulation domain into smaller chunks, with each MPI process responsible for operating on its local portion of the grid independently.

The code adopts a domain decomposition strategy, where the volumetric space is divided into smaller subdomains, each handled by a distinct MPI process. This partitioning allows for parallel execution of the simulation across multiple processes, enabling scalable performance on distributed-memory architectures.

The decomposition strategy aligns with the SPMD (Single Program, Multiple Data) paradigm characteristic of MPI applications. Each MPI process executes the same program code but operates on different data subsets corresponding to its assigned portion of the grid. This approach fosters parallelism by allowing processes to work independently on their local data without the need for coordination with other processes during computation.

The decomposition of the grid into smaller subdomains enables efficient parallelization of the simulation tasks, including cell evolution and data aggregation. By operating on smaller data subsets, each MPI process minimizes communication overhead and maximizes computational efficiency. Additionally, the use of decomposition facilitates load balancing, ensuring that each process receives a comparable workload to optimize parallel performance.

Moreover, the decomposition strategy aligns with the scalability goals of parallel computing, allowing the code to scale effectively as the problem size and the number of MPI processes increase. As the grid is partitioned into smaller chunks, the communication overhead remains manageable, and the parallel efficiency is maintained even on large-scale distributed systems.

Overall, decomposition plays a fundamental role in the MPI-based parallelization of the 3D Game of Life simulation, enabling efficient utilization of computational resources and facilitating scalability across distributed-memory architectures. By partitioning the simulation domain into smaller subdomains and distributing the workload among MPI processes, the code achieves enhanced performance and scalability for large-scale simulations.

4 Synchronization Concerns

In the MPI-based parallelization of the 3D Game of Life simulation, synchronization concerns arise primarily due to the need to coordinate access to shared resources and ensure data consistency across MPI processes. Given the distributed-memory nature of MPI programming, proper synchronization mechanisms are essential to prevent race conditions and maintain the integrity of simulation results.

One of the primary synchronization mechanisms employed in the code involves global reduction operations to aggregate data from all

MPI processes. For example, after each generation of the simulation, `MPI_Reduce` is used to compute global counts of species populations across all processes. This reduction operation accumulates counts from individual MPI processes into a global array, ensuring that the final counts are accurate and consistent across all processes.

Additionally, synchronization concerns arise when updating shared data structures, such as the grid representing the simulation domain. In the code, MPI communication primitives, such as `MPI_Send` and `MPI_Recv`, are used to exchange boundary data between adjacent MPI processes. These communication operations, such as reduction and broadcasting, ensure that neighboring processes have up-to-date information about the grid state, enabling consistent computation of cell evolution across process boundaries.

Furthermore, proper memory access synchronization is crucial to avoid data races when multiple MPI processes access shared memory locations concurrently. In the code, MPI communication primitives are used to enforce synchronization during grid updates, ensuring that conflicting memory accesses are serialized to prevent race conditions. By synchronizing memory accesses, the code preserves data integrity and prevents inconsistencies that could arise in a parallel execution environment.

Overall, synchronization concerns in the MPI-based parallelization of the 3D Game of Life simulation revolve around coordinating access to shared resources, aggregating data from multiple processes, and ensuring data consistency across process boundaries. Through the use of synchronization mechanisms such as global reduction operations and MPI communication primitives, the code achieves proper coordination and synchronization, enabling accurate and consistent simulation results across distributed-memory architectures.

5 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. Spatial decomposition divides the three-dimensional grid into smaller subdomains, with each MPI process responsible for simulating a distinct portion of the grid. By partitioning the grid in this manner, the workload is evenly distributed among processes, facilitating parallel execution without overburdening any specific process. This decomposition strategy ensures that each process operates on a comparable amount of data, promoting load balance across the parallel environment.

Additionally, dynamic workload distribution techniques are employed to adaptively adjust the computational load among MPI processes based on the complexity of the simulation. This dynamic approach ensures that processes with lighter workloads can assist those with heavier workloads, thereby preventing potential bottlenecks and minimizing overall execution time. Techniques such as task stealing or workload migration may be implemented to redistribute tasks dynamically and maintain load balance throughout the simulation.

Furthermore, the code leverages collective communication operations provided by MPI to synchronize data and coordinate parallel execution efficiently. Collective operations such as `MPI_Bcast` and `MPI_Reduce` facilitate the exchange of boundary information and global computations among MPI processes, enabling synchronized progression of the simulation while maintaining load balance.

Overall, the combination of spatial decomposition, dynamic workload distribution, and collective communication operations ensures effective load balancing in the MPI implementation of the 3D Game of Life simulation. By distributing the computational workload evenly among MPI processes and minimizing communication overhead, the code maximizes parallel efficiency and scalability, enabling optimal utilization of distributed memory systems for large-scale simulations.

6 Performance Results

The following results were obtained using the RNL Cluster.

Time	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads	32 Threads
3	585,7	300,5	152,1	77,7	40,3	27,2
10	245,9	126,4	63,5	32,1	16,3	10,5
200	68,8	34,4	17,2	8,7	4,8	2,9
1000	40,3	21,1	10,6	5,5	2,9	2,2

SpeedUp	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads	32 Threads
3	-	1,94908486	3,85075608	7,53796654	14,5334988	21,5330882
10	-	1,94541139	3,87244094	7,66043614	15,0858896	23,4190476
200	-	2	4	7,90804598	14,3333333	23,7241379
1000	-	1,90995261	3,80188679	7,32727273	13,8965517	18,3181818

Figure 1: Time Execution and SpeedUp Calculation

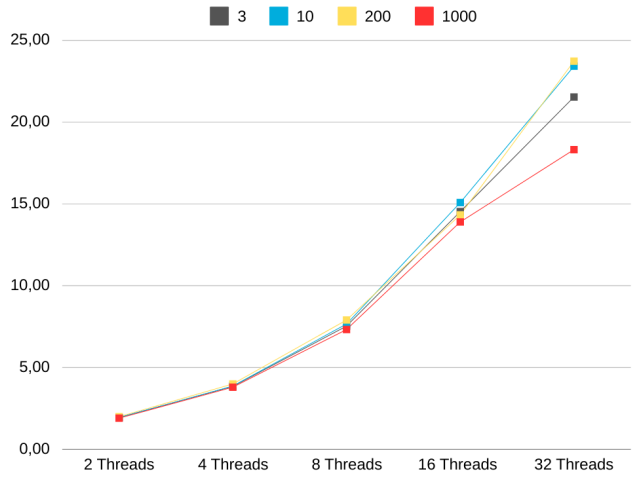

Figure 2: Line graph with SpeedUp values for each sample according to the number of threads used

After conducting performance evaluations on the MPI + OpenMP solution, the results indicate significant improvements in the performance and scalability of the 3D cellular automaton simulation. By combining power of MPI for inter-node communication and OpenMP for intra-node parallelization, the simulation achieves enhanced speedup and efficiency compared to its serial counterpart.

- Optimal Thread Count: The performance results evidence a great scalability of our solution. This is shown by the speedup almost matching the number of threads from 2 to 16 threads. This suggests that the workload is efficiently distributed among the available threads. It is worth noting that the speedup consistently remains below the number of threads employed, which is logical given that not all of the code can be parallelized. As we did not have a limitation in terms of CPU cores(the number of threads matches the number of physical cores used), we can see a gradual increment of the speedup that corresponds to the number of cores. The exception to this rule is using 32 threads, since the speedup never goes above 24. This is might be due to memory bottleneck, thread contention or simply because of the Amdahl's Law, that states that the theoretical speedup of a parallel program is limited by the fraction of the program that cannot be parallelized. Overall, we believe that the optimal thread count is 8++, because it makes use of all the cores available. Any thread count beyond this may result in higher speedups, but it won't maximize core utilization, rendering it overkill. - Speedup Analysis: Achieving a speedup of approximately 24 with 32 threads showcases a significant improvement compared to the serial CPU version implementation. This result demonstrates the efficiency of parallelization, leveraging multiple cores to execute the simulation much faster. - 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.

7 Conclusion

In conclusion, the combined use of OpenMP and MPI in this implementation underscores the versatility and adaptability of parallel computing paradigms. By leveraging OpenMP for shared-memory parallelization within individual compute nodes and MPI for communication and coordination across distributed systems, this solution maximizes resource utilization and scalability across heterogeneous architectures. This hybrid approach not only enhances performance but also offers flexibility in deploying simulations on diverse computing environments, ranging from multicore workstations to high-performance computing clusters. Continued refinement and optimization of this hybrid model hold promise for unlocking even greater computational efficiency.