# Conway's Game Of Life Simulation


1st Kevin Zheng
*Department of Computer Science*
*University of Bristol*
ad23100@bristol.ac.uk

2nd Sriram Cheedi
*Department of Computer Science*
*University of Bristol*
dz23405@bristol.ac.uk

3rd Vladimir Dementyev
*Department of Computer Science*
*University of Bristol*
ns22496@bristol.ac.uk


## Parallel Implementation

### A. *Introduction*

Multi-threading/concurrent programming makes sure that implementations stays scalable and responsive. This implementation aims to optimize the original sequential solution by dividing tasks across multiple threads, reducing computation time and improving efficiency. This section of the report details the design and implementation of the parallel approach, explores and evaluates the benchmarking results, and analyses the scalability with increasing number of worker threads.

### I. Functionality and Design

### A. *Initial Serial Implementation*

The initial serial implementation of involved a 2d slice to represent the grid, where cell states were determined by neighboring cells according to the rules in Fig. 1.

Let $S(x, y, t) \in \{0, 1\}$ be the state of cell $(x, y)$ at time $t$, where 1 represents alive and 0 represents dead. Let $N(x, y, t)$ be the count of live neighbors of $(x, y)$.

$$S(x, y, t+1) = \begin{cases} 0 & \text{if } S(x, y, t) = 1 \text{ and } (N < 2 \text{ or } N > 3) \\ 1 & \text{if } S(x, y, t) = 1 \text{ and } (N = 2 \text{ or } N = 3) \\ 1 & \text{if } S(x, y, t) = 0 \text{ and } N = 3 \\ 0 & \text{otherwise} \end{cases}$$

Fig. 1. Game of Life Rules

The serial version iterated over each cell in sequence, updating states based on the current configuration of the board.

### B. *Parallel Design and Worker Goroutines*

The implementation leverages go-routines, splitting the board into n sections, where n is the number of threads, and each go-routine processes each slice concurrently. The parallel implementation boosts the computation speed significantly compared to single thread/serial implementation.

The communication between worker threads is done through channels. In order to make sure that re-assembly of the board after worker threads process the turn comes in correct order, a slice of channels is created. Each go-routine comes with an index, which the order of receiving the processed turn is based on.

Another important aspect is edge handling, the implementation uses a torus topology through the modulus operator to enable the cells to "wrap around" and interact with cells on the opposite edge. This allowed each goroutine to access edge neighbors without additional synchronization overhead, as each cell's neighbor interactions are resolved within local slices.

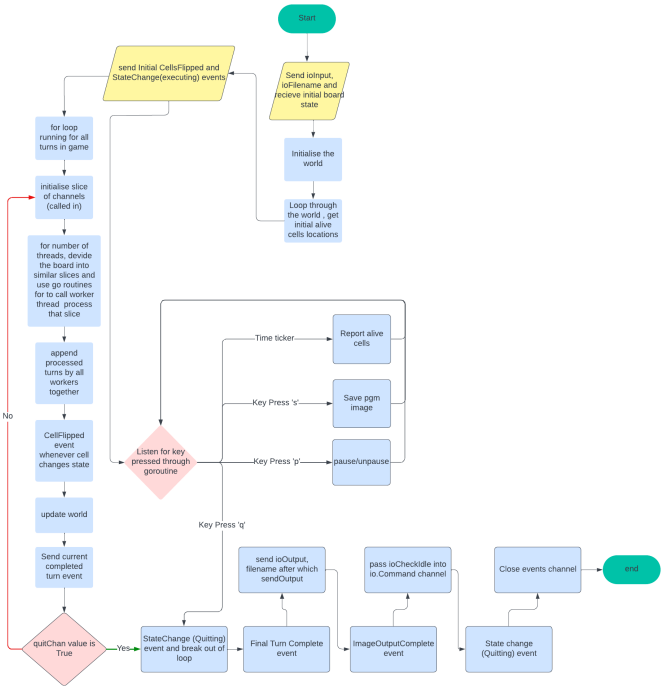


Fig. 2. Flowchart For Parallel Implementation

Further specification of simulation process flow can be seen in Flowchart depicted in Fig. 2.

### C. *Ticker and Keyboard Goroutines*

The implementation includes ticker that reports the count of alive cells every 2 seconds. The ticker uses separate go routine to avoid any potential interference with the rest of the simulation and main simulation loop. It receives the current turn number and world matrix from channels updated by the main simulation loop. After each turn, the main loop sends this data, allowing the ticker to calculate and report alive cells without disrupting ongoing computations. Additionally, there are 2 dedicated channels that manage communications between the main loop and the keyboard presses. These channels handle user inputs related to pausing and quitting.

## II. Benchmarking and Performance Analysis

### A. Methodology

To evaluate the performance of our program, we conducted a series of benchmarks measuring the time required to iterate over 1000 turns of the Game of Life simulation.

Each test was repeated at least 6 times to minimize the influence of random fluctuations in performance, ensuring that the results reflect the true efficiency of the program rather than temporary anomalies. All measurements were collected on an Intel(R) Core(TM) i7-14700 processor, providing a consistent hardware environment for the tests.

This methodology ensures that the benchmark results are reliable and not significantly affected by variations in hardware performance, offering a clear representation of the program's efficiency under controlled conditions through graphs and evaluation.

### B. Analysis of Thread Efficiency Across Various World Sizes

Fig. 3 illustrates the runtimes across varying thread counts. Processing is least efficient in the single-threaded mode, where all computation is handled by a single thread, resulting in extended execution times, particularly for larger images. Specifically, the runtime for a single thread is 8.48 seconds per operation, whereas utilizing 16 threads reduces the runtime to 1.89 seconds per operation — a 4.48x improvement.

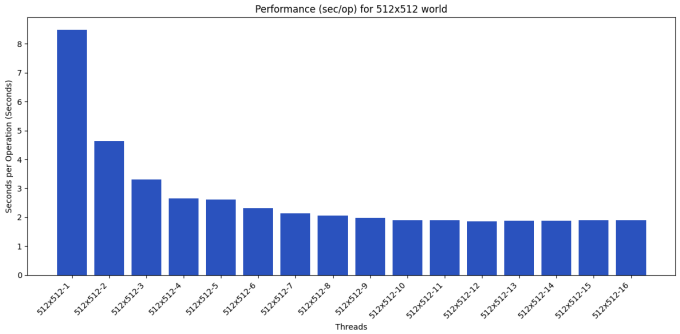


Fig. 3. Benchmark - Time per Operation for the parallel implementation for 512x512

For 2-4 threads, the performance is high, as multiple threads can execute tasks in parallel across available CPU cores. When the thread count increases from 5 to 16, the performance continues at a decreasing rate. This is because of the synchronization overhead as more threads used.

The efficiency of multi-threading in processing Game of Life simulations varies significantly depending on the size of the game world. For smaller game worlds, such as 16x16 shown in Fig. 4, the addition of more threads does not provide a substantial improvement in processing efficiency. This is due to the relatively low computational demand of the task, where the overhead of managing multiple threads outweighs any performance gains from parallel execution.

In contrast, for bigger worlds (e.g., 512x512 shown in Fig. 3 and 64x64 shown in Fig. 5), the parallel processing begins to show its advantages. As more threads are employed, the time decreases gradually, showing that multi-threading is effective in handling larger datasets. This highlights the scalability of the approach, where the benefits of parallelism become increasingly apparent as the size of the problem grows.

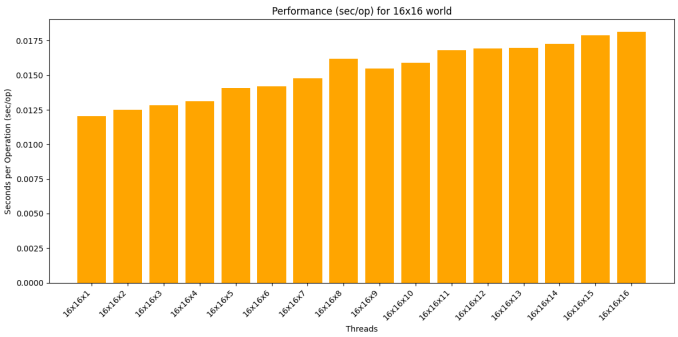


Fig. 4. Benchmark - Time per Operation for the parallel implementation for 16x16

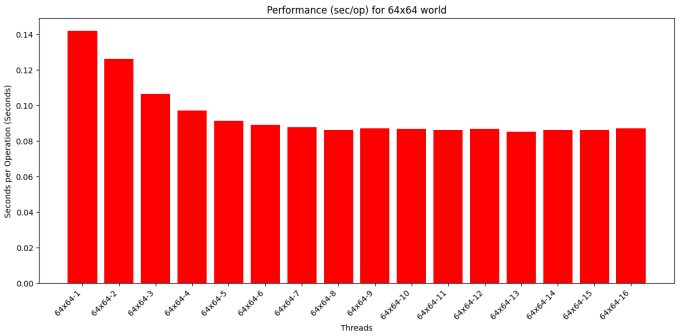


Fig. 5. Benchmark - Time per Operation for the parallel implementation for 64x64

### C. CPU profiling

Fig. 6 suggests that the calculateNextState function from gol package accounts for the biggest portion of the runtime, approximately 255720 milliseconds, 70.14 percent of the total computation time. This computationally intensive function is called every iteration, iterating through each cell in the board and applying the Fig. 1 rules based on cell's neighbors, returning the new board.

The distributor function itself introduces overhead by coordinating data exchange among threads, as it accounts for 3.68 percent of the whole CPU run time. Although its contribution is much less compared to calculateNextState function, optimization done to the orchestrating mechanics can improve the run time.

Since the calculateNextState function consumes majority of the CPU time, optimization in this function would result in substantial improvement in runtime. Optimization ideas can be found in Potential Improvements section, next section of this report.

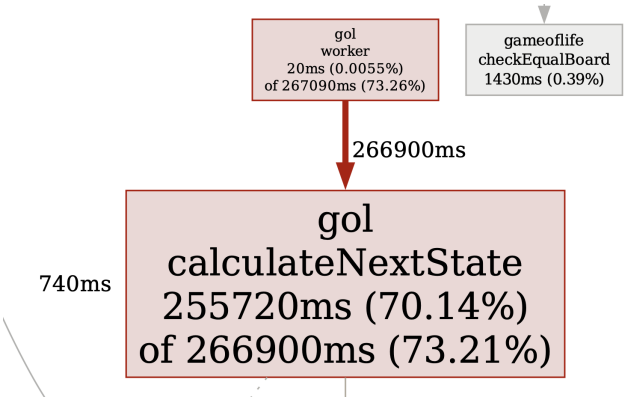


Fig. 6. CPU profile of machine running the benchmarks

## III. Potential Improvements

### A. *Potential Improvements*

Firstly, as the number of goroutines increases, channels can introduce latency due to overhead passing. The shared memory can potentially reduce the communication overhead. The implementation may use synchronization mechanisms like mutexes and semaphores to control the access to shared regions of the board. Therefore, instead of passing the data back and forth through channels, workers would be able to directly update shared memory, most likely significantly improving memory usage and computation time.

Secondly, the modulus operator can be computationally expensive, especially when the scale of the program increases, performed frequently. Ghost cells or duplicated edge rows/columns could avoid modulus calculations entirely. The solution can be implemented through addition of 2 extra columns on each side of the board to store edge values, eliminating usage of modulus operator for boundary checks, reducing computational speed.

## IV. Conclusion

In summary, this parallel implementation of Conway's Game of Life shows gain in performance and scalability with larger boards and more threads. From the analysis of the benchmark results, it clearly demonstrates that the performance increases as the threads increase for larger boards, but is proven otherwise on smaller worlds.

# Distributed Implementation

## I. System Design and Functionality

### A. *Single Threaded Design*

The initial distributed system utilizes a single worker/server paired with a distributor/controller to implement a distributed version of the Game of Life simulation (GOL). The design aims to decouple the GOL computation from the controller and delegate it to a worker, and define requests, response structures and functionalities in a stubs file to enable structured communication and provide a layer of abstraction between the controller and the worker.

The system workflow begins with the distributor receiving the initial world state via an I/O operation. This state, including the Game of Life grid parameters, is then sent to the worker through a Remote Procedure Call (RPC). Upon receiving the initial data, the worker performs all turns of the Game of Life computation in the function *CalculateGameOfLife*. After completing the simulation, the worker returns the updated world state to the distributor via RPC. See Fig. 7 for the system design.

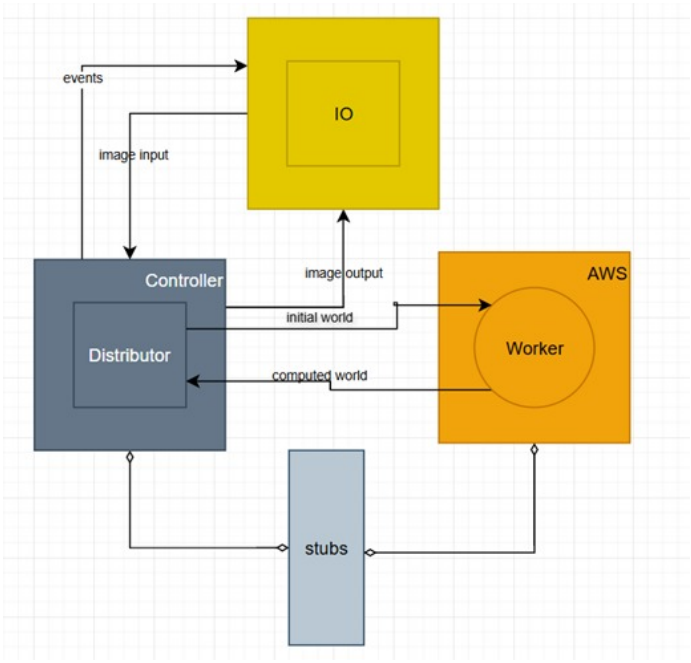


Fig. 7. Single Threaded Design

### B. *Introduction of broker and multiple workers*

In this distributed system, a broker component is introduced to manage the communications between the controller and workers. The distributor fisrt sends the data to the broker via RPC call. After receiving this data, the *CalculateGameOfLife* function in the broker divides the world grid into four sections and initiate concurrent RPC calls to four workers to calculate the next state in the function of *CalculateNextState*. Once the workers have computed their assigned world slices, the broker merges the results and reconstructs the world for the

next iteration. Within the broker, certain shared resources in a *GolOperation struct* are leveraged to ensure smooth communication and synchronization between the controller, broker and workers. See Fig. 8 for the system design.

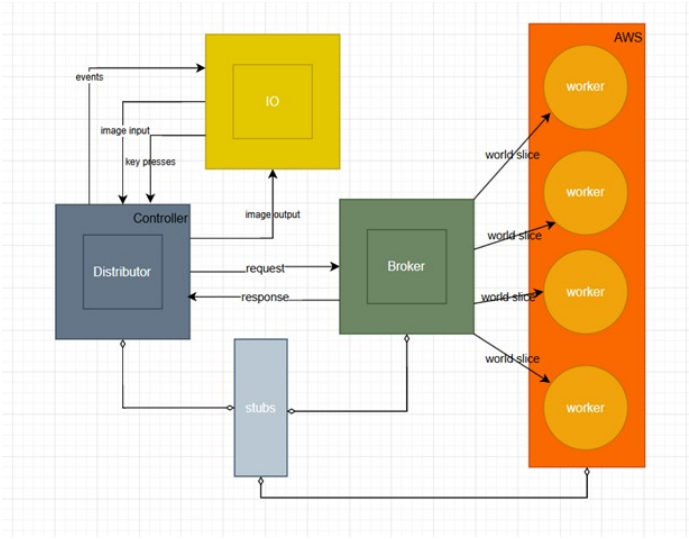


Fig. 8. Broker and Multiple workers design

### C. Potential Scaling and Pitfalls of the Distributed System

**Increased Parallelism:** The system can scale with more worker nodes by further divide up the world slices. This can make the system more efficient and adaptable to larger world slices.

**Data redundancy:** Multiple brokers can be introduced to enhance the robustness of the system. Crucial data, such as world and turns, can be copied across brokers to prevent a single point of failure. If the primary broker fails, other brokers can take over the unfinished task.

**Component Failures and Network Dependency:** The system's dependence on stable network connectivity introduces critical points of failure. Our system is robust to the disruption of network between the controller and broker due to fault tolerance. However, the data integrity between the broker and workers can still be compromised since there is no network error handling mechanism between them. When individual worker is disconnected from the broker during computation, the world returned will be inaccurate.

### D. Alive Cells Count implementation

In the distributed system, a ticker mechanism is employed to periodically make an RPC call in the controller to the broker to inspect the number of alive cells and current turn of the Game of life computation. The call occurs in a separate goroutine to prevent blocking of the other execution in the controller. Within the broker, the function responsible for reporting the number of alive cells and turns must first acquire a mutex lock to prevent race conditions and send the data back to the controller in response.

### E. Keyboard Press implementation

Within the controller, the key press channel is placed within the same select statement as the ticker channel. Each key press sends an RPC request to the broker, which then in the function *KeyPressHandle* processes the specific instruction associated with each key.

**'s' Key (Save State):** The broker acquires a mutex lock and responds to the controller with the current world and turn data. The controller then saves this state as a Portable Gray Map (PGM) file.

**'q' Key (Quit Simulation)**: An RPC request from the controller is sent to the broker to halt the Game of Life simulation using the function *CalculateGameOfLife*. The broker acquires a lock and sets the **quit flag** of *GolOperation struct* to false, signaling the main computation loop to halt any further calculations and causing the *CalculateGameOfLife* function to enter an idle loop. It then sends the current world and turn data back to the controller for it to save the final world state as a PGM file.

**'k' Key (Systematic Shutdown)**: Shut down for all components of the Game of Life simulation in the following order: controller, broker, and worker servers. The rationale behind this order comes from the fact that each component needs to signal the next in the RPC requesting chain to ensure proper shutdown of the system. After receiving the 'k' key press, the controller sends an RPC request to the broker, signaling it to stop the simulation. The corresponding function in the broker then acquires a mutex lock and sets the **quit flag** of *GolOperation struct* to true, indicating that the game calculations in *CalculateGameOfLife* should stop. Following this, the broker initiates a goroutine that sleeps for one second before exiting, allowing sufficient time for controller to shut down and concurrently sending RPC requests to each worker server to terminate their processes.

**'p' Key (Pause/Resume Simulation)**: The controller sends an RPC request to the broker to pause the processing of GOL calculation. The function in broker first acquires the mutex lock and responds with the current simulation turn and updates the **pause flag** accordingly. Upon receiving this data, the controller places a *Paused* event in its event channel. Pressing the 'p' key again will trigger another RPC request to the broker to resume the process. Similar to the pause action, the broker sends the current turn back to the controller, and the controller updates its event channel with an *Executing* event to signal that the simulation has resumed.

### F. Extensions

*1) **Parallel Distributed System**:* The computation in the distributed system can be parallelised to make the computation

more efficient. After receving the assigned world, the workers further divide the slices into smaller segments specified by the number of threads in the request.

*2) **SDL Live View of Distributed Implementation**:* In a separate goroutine within the controller, a request is sent to the broker each turn to acquire the previous and current world, as well as the current turn. The corresponding function in the broker then awaits data from its channels, which receives updates from the Game of Life (GOL) computation loop. Once the data is available, the broker responds to the controller with the data. The controller then calculates the flipped cells and sends a flipped cells event and turn complete event to the event channel.

*3) **Fault Tolerance**:* At the beginning of the execution of the distributor, a request is sent to the broker to inspect the status of the continue flag, which is set to false for the first client. Once 'q' key press is triggered, the broker will set the **continue flag** to true. Consequently, when a subsequent client establishes a connection to the broker, the initial RPC response provides a true continue flag, along with the previous world state and the turn count from the last completed iteration, which updates the parameters of the GOL computation RPC call, enabling the GOL computation to be reinitialized with the prior session's state, allowing the computation to proceed seamlessly from the last recorded turn.

## II. Benchmark Analysis

### A. Methodology

To evaluate the performance of our program, we conducted a series of benchmarks measuring the time required to iterate over 100 turns of the Game of Life simulation. The turns are limited to 100 due to the TCP communication overhead.

Each test was repeated at least 6 times to minimize random noise, ensuring that the results reflect the true efficiency of the program rather than temporary anomalies. We ran the broker and the workers on large AWS instances and the distributor on our local machine.

### B. Analysis

*1) **Distributed-System**:* The benchmark results for this implementation can be seen in Fig 9. We can notice that the overall run time decreases by having fluctuations between other threads. It is suspected that the fluctuations come from the introduction of RPC calls.

*2) **SDL vs No SDL**:* The benchmark results of Parallel-Distributed System with SDL and without SDL can be seen in Fig.10. The overall runtime for Parallel-Distributed System with SDL is longer compared to Parallel-Distributed System without SDL due to the overhead caused by the RPC calls and channel communications. Specifically, the SDL function in the controller has to send a request for every turn to the broker to obtain the data needed and send to the "CellsFlipped" event channel. Additionally, the function inside the broker acquires the data through channel communication with the main computation loop.

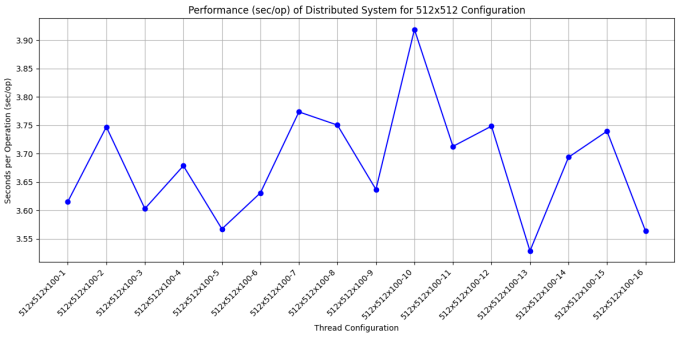


Fig. 9. Benchmark- Time Operation for the Distribution Implementation for 512x512

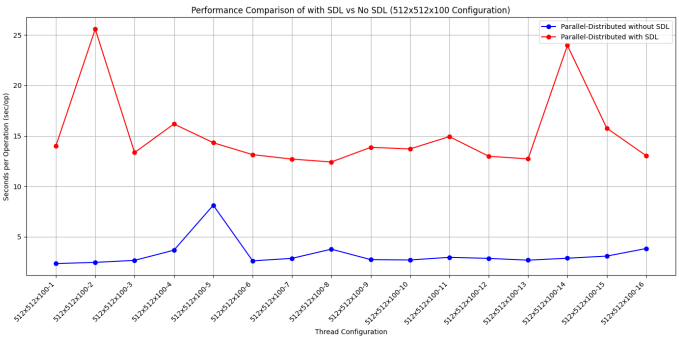


Fig. 10. Benchmark- Time Operation for the SDL vs No SDL for 512x512

## III. Potential Improvements

### A. Potential Improvements

Firstly, the implementation creates a "bottleneck" and potential latency in data exchanges from each worker resynchronizing, sending and receiving data through the central distributor, and as grid size and number worker increase, it may introduce significant communication overhead. To solve this issue, a more decentralized communication system can be used, such as halo exchange, enabling workers to interact with few immediate neighboring worker rather than central hub, therefore improving scalability, reduces latency in data propagation and reducing communication overhead.

Secondly, adjusting thread workloads dynamically for more equal workloads. The last thread might get the least workload as it processes the remaining of the board, potentially resulting in the CPU either under-utilized or over-utilized, therefore affecting the computation time negatively.

Lastly, current implementation heavily relies on mutex locks, which may cause increased lock contention, the solution to the problem may be to utilize read-write locks or optimistic concurrency control, further improving parallel efficiency especially in high-workload situations.

## IV. Conclusion

In summary, the distributed system designed and implemented is scalable and robust in certain situations. But certain improvements can be made to enhance the scalability and robustness of the system, such as data redundancy and halo exchange. With regard to performance, the introduction of parallelism to the multiple distributed system improves the computational efficiency, whereas the introduction of SDL view decreases it due to communication overhead.