Parallelised and Distributed Implementations of Conway's Game of Life

Philip Mortimer (ia21629@bristol.ac.uk) and Beatrice Jiang (ow21096@bristol.ac.uk)

Abstract

We have made use of a wide range of concurrent programming techniques to produce fast algorithms to simulate Conway's Game of Life in Golang. We concluded that a pure memory sharing algorithm that divides the GoL (Game of Life) grid into chunks for workers to process performed best. We created a distributed system that made use of a broker and a number of AWS servers to efficiently compute GoL.

1 - Parallel Implementation

To form a strong basis for our parallel implementation, we produced a naïve algorithm that stores the board in a 2d slice of bytes. We simply iterated over each cell and applied the Game of Life update algorithm.

In our version of Game of Life, the grid is considered to be toroidal. This allows us to divide the grid into independent chunks that different threads can tackle simultaneously. We divided the grid into chunks by assigning each thread a number of consecutive rows to process. Our function, *divideGridForProcessing*, takes in the height of the grid and the number of threads available and assigns each process a number of rows to deal with such that all threads deal with the same number of rows (with some threads potentially dealing with one extra row).

1.1 Benchmarking Methodology

Below, we discuss the development cycle for our parallel algorithm. In order to benchmark our solutions, we measured how the number of threads impacts the time to compute the final grid state. We used a 512x512 image for 1000 turns and repeated each measurement ten times to reduce the effect of noise. We benchmarked our solution on Linux lab machines, which have an i7 chip and 6 cores (which equates to 12 threads with hyperthreading). We measure the performance of our system for 1—16 workers.

Additionally, to measure the performance difference when the Game of Life GUI is enabled, we measure the time taken to process 10000 turns with 8 threads on a 512x512 image. This is repeated numerous times to reduce noise and is also performed on the lab machines.

1.2 Ghost Rows with Repeated Memory Allocation

The difficulty with our parallel implementation arises when a worker processing a chunk of the grid has to access the state of cells outside of their chunk. Our initial solution passes two ghost rows to each worker, as well as their chunk of the grid. The ghost rows represent the list of cells directly above and below the chunk. Once every worker has evolved the grid by one turn, the new grid and ghost rows are created. This approach requires us to allocate memory for all the ghost rows and the new grid for every single iteration of evolution.

1.3 Ghost Rows with No Memory Allocation

Repeated memory allocation is unnecessary and adds significant overhead to the efficiency of the algorithm. Thus, we create a *currentWorld* and *nextWorld* structure at initialisation. The idea is that each worker reads from *currentWorld* and writes the updated state to *nextWorld*. At the end of each turn, the pointers are swapped so that *currentWorld* points to the updated game state and

nextWorld points to the state of the previous turn. This pointer swap is done in O(1), allowing us to save repeated memory allocation.

1.4 Optimised Parallel Algorithm without Ghost Rows

Due to the enhancements made by removing repeated memory allocation, we have a structure which we only ever read from when evolving the grid, and one which we only write to. Each cell in the new structure is written to exactly once (by exactly one thread). Thus, we can safely use the two as (effective) global variables and have threads read from *currentWorld* and write to *nextWorld* without having to worry about synchronising access between threads. This makes our code much more readable and cleaner. Avoiding having to create and keep track of ghost rows should produce significant performance boosts. One of the largest costs in the program is the function *cellAlive* which takes in a grid and a set of coordinates and returns 1 if the cell at the provided location is alive and 0 if not. This function is called 8*width*height*turns times. Significant speed improvements are achieved by not having to check whether a ghost row should be accessed. Whilst we believe that speed up is achievable from this algorithm, we would argue that this is a well-optimised channel-based implementation for the parallel section.

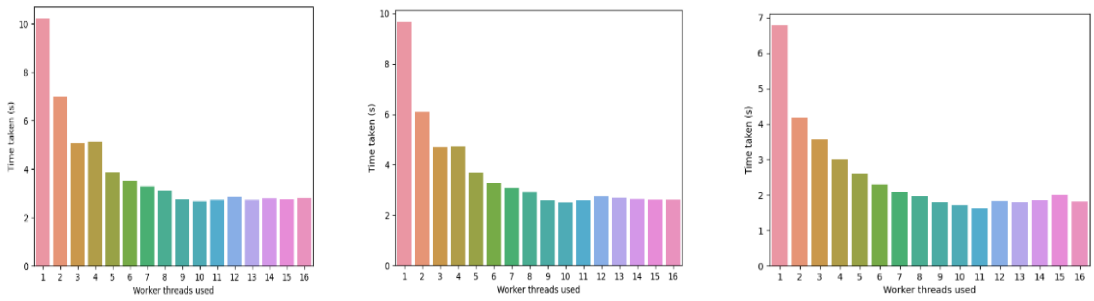


Figure 1 – (Left to right) Performance with repeated memory allocation, without repeated allocation, without repeated allocation and without ghost rows.

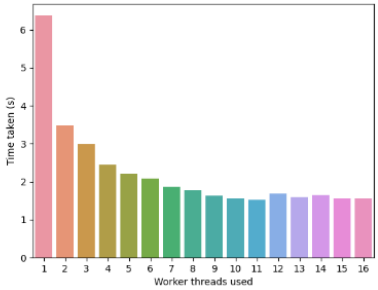
1.5 Memory Sharing

To speed up our algorithm, we decided to remove channels from our parallel implementation and use more traditional synchronisation mechanisms such as semaphores, mutex locks and conditional variables. Firstly, we removed the use of any channels between the distributor and the workers. Fortunately, we designed the algorithm to inherently make use of memory sharing and thus did not need to change too much of our algorithm.

Next, we replaced IO communication with pure memory sharing. To achieve this, two goroutines within the IO channel listen for write and read commands using two semaphores per routine. Additionally, a mutex lock is used to signal when the system is idle / is not idle. Each time an IO command is sent or received, a shared grid is accessed and either written to or read from. Our analysis indicates that this achieved a minimal speed up which we presume amounts to noise. We believe this is likely because IO actions are performed so infrequently, that any speed ups are insignificant.

Next, we removed the channels that are used to manage key presses and events. This resulted in a small but noticeable improvement in benchmark results. However, it sped up GUI processing by about 2.7 times. We believe that this is due to the way event processing is handled. Specifically, the handling of flipping a cell is far faster as each thread is able to safely flip a cell without having to send an event. Instead, each worker simply flips the appropriate pixels in the window and continues

processing. Once the turn complete event is sent, the image is redrawn. This allows concurrent pixel flipping. It also avoids underlying channel synchronisation methods that add bloat when sending an event through a channel. This optimisation is significant as the cell flipped event is called within the critical loop of the worker.



Algorithm	*Average Process Time with SDL Window (s)*	*Relative Speed*
Channel Based	169	1.0x
Pure Memory Sharing	63	2.7x

Figure 2 – Pure Memory Sharing Benchmark

1.7 Benchmark Analysis

Algorithm	*Relative Speed (1 Thread)*	*Relative Speed (8 Threads)*	*Relative Speed (16 Threads)*	*Relative Speed (Average across 1-16 Threads)*
Ghost Rows with Repeated Memory Allocation	1.0x	1.0x	1.0x	1.0x
Ghost Rows with No Memory Allocation	1.1x	1.1x	1.1x	1.1x
Optimised Channel Based	1.5x	1.6x	1.6x	1.5x
Pure Memory Sharing	1.6x	1.8x	1.8x	1.8x

Our parallel implementations scale well with the number of threads, with execution speed rapidly increasing between 1 and 2 threads. As more threads are added, the performance increases, though the rate of improvement slows down until it plateaus at about 11 threads. We speculate that this occurs because of the physical limits of the benchmarking device which has 6 cores. Using hyperthreading, this equates to roughly 12 logical threads. Thus, we would expect the limit to be at around 12 threads. In an ideal situation, the execution time would halve with double the number of threads added until 12 threads. However, hardware bottlenecks, alongside the need to synchronise the world state after each evolution (and at other points such as keyboard presses) mean that this does not occur in practice. Despite this, our solution is clearly well parallelised as it scales effectively with the number of cores. Our solution tends towards optimal parallelisation as the size of the input grid increases.

Unsurprisingly, our optimised channel-based solution and pure memory sharing solution outperform ghost row-based solutions. The performance differential between algorithms appears to remain constant regardless of the number of threads, showing that our algorithms scale nicely.

1.8 Future Improvements

In future, we would look to optimise the critical loop of each worker. We would achieve this by making more use of global variables and less use of parameter passing. For example, the *cellAlive*

function takes a grid as input every time it is called. Given how frequently it is called, it would be beneficial to avoid this. However, improvements achieved by such optimisations would be minimal.

For significant optimisation, we would look to represent the board as a sequence of words, using one bit to represent one cell (currently one byte is used per cell). Using fast bitwise operations, we could update 64 cells at once. Further speed could also be obtained by keeping track of which cells have been changed in the last iteration and only updating these cells and their direct neighbours. Both approaches have been shown to be effective (Oxman et al 2013) (Abrash 1997) (Finch 2003).

2- Distributed Implementation

In addition to a parallel solution, we sought to create a distributed solution that supports multiple worker nodes communicating to a controller over a network connection. This allows us to scale the processing of the Game of Life beyond the reasonable limits of a single computer. It also allows us to interact with the Game of Life locally without requiring knowledge of how to implement it ourselves.

2.1 Distributed System to Calculate Final Grid State

For simplicity, we started off with the aim of having a single worker node computing the whole Game of Life and returning this result. The local controller initiates this process by making an RPC connection with the worker. The distributor communicates with IO and events channel in a similar fashion to the parallel solution. However, we do not have to send cell flipped notifications after every turn has been processed. This is an important difference to the parallel solution as we have shown that flipping cells is a bottleneck in the overall runtime.

In order to realise the full potential of a distributed solution, we need the controller to be able to make use of an arbitrary number of worker nodes. In order to effectively control this, we introduced a broker. This is hugely beneficial as having the controller handle communication with all the servers would have been very tricky and convoluted. Instead, the controller is provided a simple black box interface through which they may interact with the broker.

The broker maintains a list of addresses of servers that it can connect to. It forms an RPC connection with each server. Upon being asked to process the grid by the controller, it divides the grid into chunks (much the same way the parallel algorithm did). Each server concurrently processes their chunk of the board and returns the grid evolved by one iteration. This approach also requires each worker to receive the above and below ghost rows. The broker does this until it has simulated every turn and then returns the list of alive cells to the controller. Through the use of flags in the broker (and a constant string in the distributor storing the IP address of the broker), we can compute Game of Life using an arbitrary number of workers.

2.2 Distributed System with User Input

To improve on our design, we sent a greater number of events to IO. We write a PGM file corresponding to the final grid state to IO, and also report the number of alive cells every two seconds with a ticker. To achieve this, the local controller sends a request for the number of alive cells every two seconds. Our primary processing loop is used in a go routine by the controller. However, simpler commands like getting the number of alive cells are called by a blocking RPC call. Therefore, we need to introduce synchronisation mechanisms to ensure that each thread within the broker safely accesses the world state. To achieve this, we lock access to the global variable that stores the current grid state whenever we update it or read from it (which happens, for example, when we calculate the number of alive cells).

We also added support for receiving user input via key presses. This allows users to pause processing, quit the controller, kill all distributed elements, and save the current state. Pausing and quitting are achieved by storing global Boolean variables in the distributor that indicate the current state of processing. All this provides us with a strong solution. Our analysis indicates that this system scales well with the number of AWS workers, in a similar fashion to our parallel system.

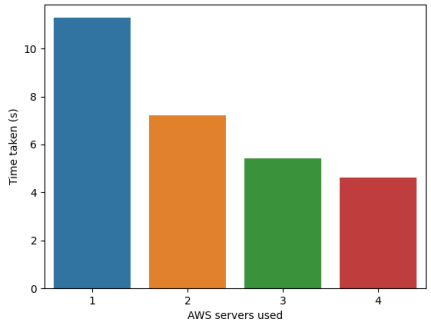


Figure 3 – Scaling of distributed system

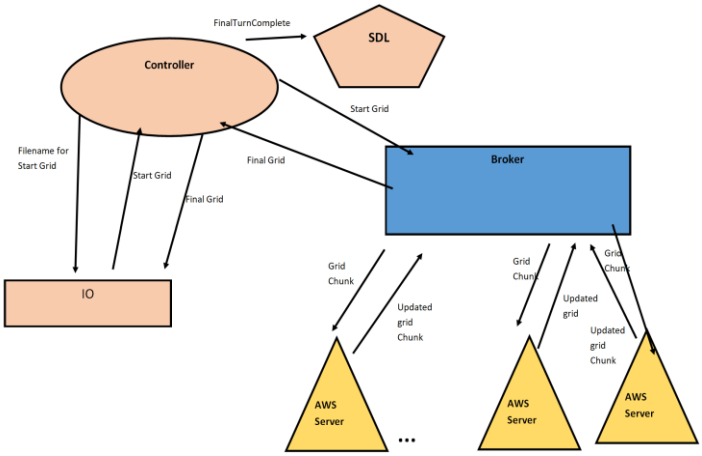


Figure 4 – Design of distributed system to calculate final GoL state

2.3 Fault Tolerance

When the user quits the controller, all computation performed by the broker is lost. As a form of fault tolerance, we have added support for having the broker resume computation from where it left off if a new controller connects with the same starting grid. This can be disabled or enabled via command line argument to the broker. For future improvements, we would suggest adding further fault tolerance features that enable the broker to deal with the failure of AWS servers by redistributing the load amongst working servers.

2.4 SDL Live View

The distributed system gains a lot of speed by not having to synchronise the grid state with the controller after every turn of processing. However, this prevents the user from having a live view of the SDL game. Thus, we established a bi-directional RPC connection between the broker and controller to facilitate this. At the end of every processing turn, the broker transmits the list of flipped cells to the controller, which in turn sends them to SDL. This allows live visualisation but comes at a great performance cost. Sending data over an RPC connection is extremely slow due to transmission costs. To minimise this, we only send the list of flipped cells once per turn. Testing indicates that having the GUI live view is about 1.9 times slower than without it. However, due to the volatility of network connections, this figure is likely to be subject to large variation.

2.5 Parallelised AWS Nodes

To improve the performance of each AWS worker, we implemented our parallelised algorithm for each AWS worker. As proven in this report, parallelisation does improve performance significantly. However, we found moderate improvements when testing this solution. Greater performance gains were achieved when testing our solution on a 5120x5120 image. We speculate that this is because the overhead required to create each thread is more significant when each worker has a negligible amount of work to do. However, calculating precise the performance difference was difficult as the reliance on data transfer speeds between nodes led to variable results.

2.6 Future Distributed Improvements

We are pleased with our distributed system and all of the extensions completed that enhance its functionality. For future improvements, we would look to implement a halo exchange scheme. Such a scheme mitigates the need for the broker to synchronise with the servers after every single turn. It also reduces the data sent via a network connection, which we believe is the single biggest bottleneck for the speed of execution. Such a system would involve each worker waiting to receive a ghost row from two other workers before processing a new turn.

Currently, our system uses a slice of bytes to represent the grid. In future, we would look to explore using bit packing to represent the grid, which would allow us to send an eight of the data. This would result in large performance gains.

Controller -> Broker

Function	Input	Return
Initialise Broker	Start Grid	Nothing
Process Game of Life	Nothing	Complete Grid
Get Alive Cells	Nothing	Alive Cells
Set Paused State	isPaused	Nothing
Quit Controller	None	None
Kill Distributed	None	None

Broker -> Server

Function	Input	Return
Evolve Grid	Grid Chunk + Ghost Rows	New Grid
Kill Server	Nothing	Nothing

Broker -> Controller (SDL Live View)

Function	Input	Return
Turn Processed	Alive cells + Turns Complete	Nothing

Figure 5 – Distributed RPC commands

Conclusion

We created numerous implementations of Conway's Game of Life that utilise parallelisation and distributed systems. Our solutions are fully functional and scale effectively. We have completed a wide range of extended tasks and have produced a strong range of solutions.

Sources

Oxman et al (2013) Computational methods for Conway's Game of Life cellular automaton, Journal of Computational Science, https://www.sciencedirect.com/science/article/pii/S187775031300094X

Abrash (1997) Graphics Programming Black Book Special, Chapter 17 + 18, https://www.jagregory.com/abrash-black-book/

Finch (2003) An implementation of Conway's Game of Life, https://dotat.at/prog/life/life.html

Submission Details

In addition to our submission, we have developed multiple other parallel and distributed versions (as discussed in this report). To access these versions, please follow this OneDrive link: https://uob-my.sharepoint.com/:f:/g/personal/ia21629_bristol_ac_uk/EkQ57TiXNXJLq9rUe0C4fIQB_SFEJ6nCmcI8N62Ch31XUA?e=5LiZ2z (expires 30/12/2022). Alternative google drive link: https://drive.google.com/drive/folders/1e9rhfIIvLktEKsz-szy3wete4rXf_zX3?usp=share_link

If this does not work, please email us for access.