Parallel and Distributed Game of Life Simulation

Aristotelis Kostelenos (gp19712@bristol.ac.uk), Paolo Mura (rk19763@bristol.ac.uk)

1 Parallel Implementation

We have created a parallel implementation of the Game of Life (GoL) in GO. It uses between one and sixteen worker goroutines to calculate the next state of the board as well as a ticker goroutine for printing the number of alive cells every two seconds. A single distributor goroutine coordinates these interactions.

1.1 Simulation Features

Before starting the simulation, the number of threads can be passed as an input parameter that determines the number of workers created by the distributor.

During the simulation, the user can pause by pressing “p”, resume by pressing “p” again, save the image with “s” and quit the simulation with “q”. The distributor receives the key presses from a channel and handles the logic in a switch statement. The SDL visualisation also displays the state of the board in real time. The `c.events` channel is passed to the workers, which send a cell flipped event whenever it is appropriate.

After the simulation is run for the number of turns specified, the final image is exported as a PGM file.

1.2 Safe Memory Sharing Approach

We decided to implement a memory sharing approach in order to optimise time and space efficiency. This solution reduces time spent copying and transferring slices since all operations are performed on the same memory.

Each worker is tasked with processing a specific set of rows from that grid. Each worker is allocated an equal number of rows. The remainder is then split between them as illustrated in Fig. 1.

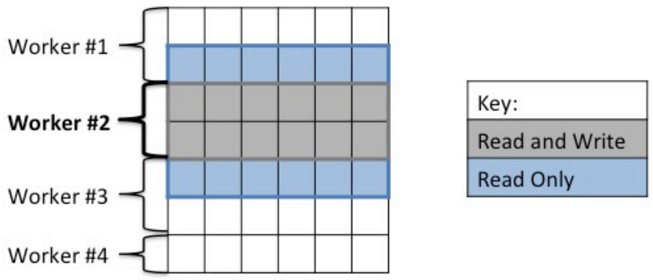

Fig. 1: Row allocation between workers, with the rows accessible to *Worker #2* highlighted.

The workers read and update the GoL board using safe memory sharing. The distributor passes the workers two pointers to slices of type `[][]uint8`. One of the slices contains the state of the board in the previous turn. The workers write the next state of the board in the other. Each worker only writes to the part of the board that it has been assigned, preventing race conditions. When reading, however, it also reads one row above and one below its assigned part. In this way, it can read from the neighbours of all its own cells. The rows a worker accesses are illustrated by example in Fig. 1 where *worker #2* is allocated two rows and also has read-only access to the row above and below.

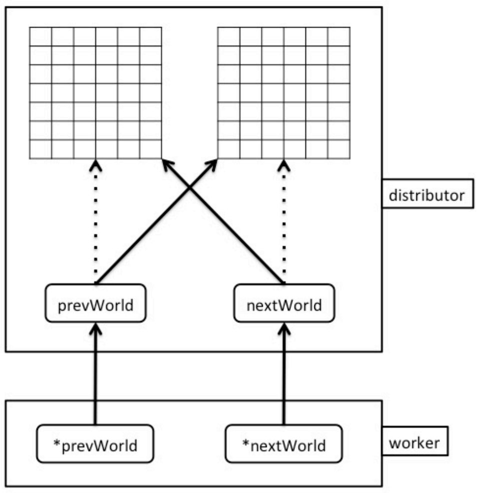

Fig. 2: Using pointers to shared memory.

The distributor receives confirmation from each of the workers via a binary semaphore that they have finished the current turn. Then it switches the two slices around as illustrated in Fig. 2. As a result of the switching, the slice the workers write to in one turn is the slice they read from in the next. We opted for this approach in order to minimize initialising new matrices and copying from one matrix to another. With each worker having its own binary semaphore, the distributor receives from all of them before switching and then posts to each of them afterwards. This ensures that the workers and distributor remain synchronised and that all the workers have finished processing before the change of the board.

1.3 Testing and Analysis

To measure the performance of our implementation, we ran several benchmarks with varying numbers of worker goroutines as well as differing board sizes and total turns.
We also measured performance in several computers with different core and thread counts.

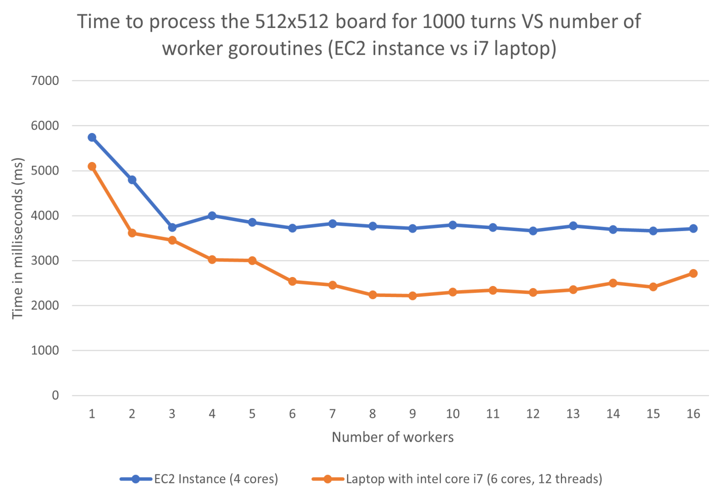

Fig. 3: Benchmark results measuring effectiveness of parallelisation in machines with different numbers of cores.

As can be seen in Fig. 3, the performance gain of parallelisation is dependent on the number of cores and threads of the machine. On the 12-thread laptop running the benchmark, when using three threads instead of one, it is 1.5x faster whereas when running with eight it is 2.3x faster. On the 4-core EC2 AWS instance, both the three- and eight-core runs produced a result that was 1.5x faster than the single worker run. It is apparent that when the number of workers nears or exceeds the number of threads of the machine, adding more workers does not improve performance.

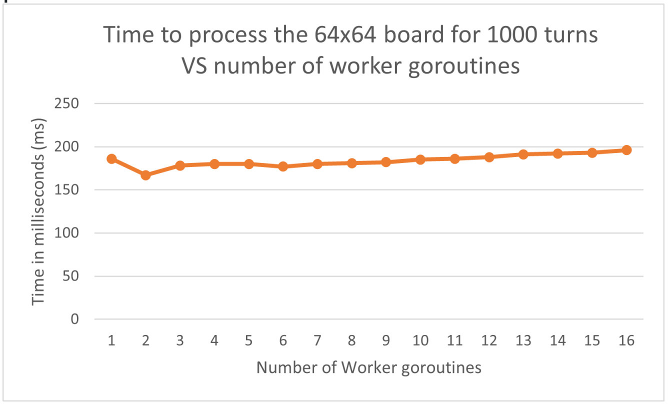

Fig. 4: benchmark results for a small board.

The gains of having many workers are also negligible when given small images. The 64x64 benchmark on the EC2 instance (see Fig. 4) showed only slightly faster times when using a few workers instead of one, but progressively worse times as more workers were added.

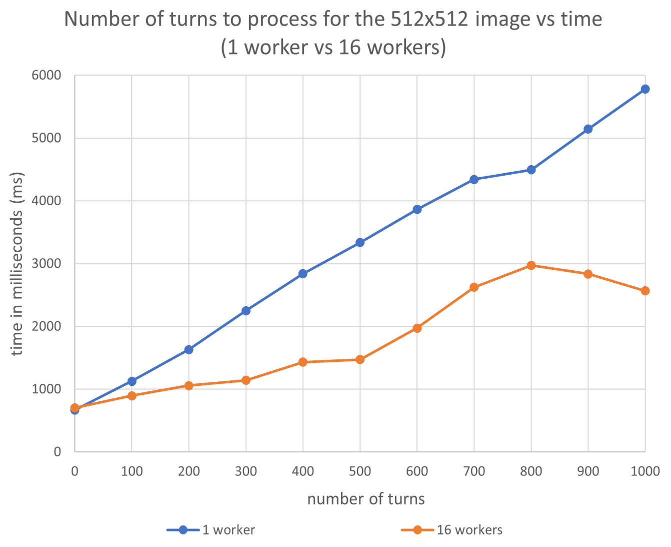

Fig. 5: Benchmark results comparing runtimes for 1vs16 workers.

Through benchmarks we confirmed that the constant amount of time needed for opening and saving a board is significant. This can be seen by the time delay in Fig. 5 when 0 turns are processed. This means that the parallelised implementation does not offer significant advantages for small numbers of turns. When measuring the efficiency of the parallelisation itself, that constant time should ideally be subtracted from the results (see Fig. 6). As an alternative, running the simulation for many turns also shows fairly accurate results as the amount of time needed for the GoL logic keeps increasing. In Fig. 5, the 16-worker run takes just as much time as the single worker one for 0 turns. It is, however, 1.25x faster when calculating 100 turns and 2.3x faster for 1000 turns.

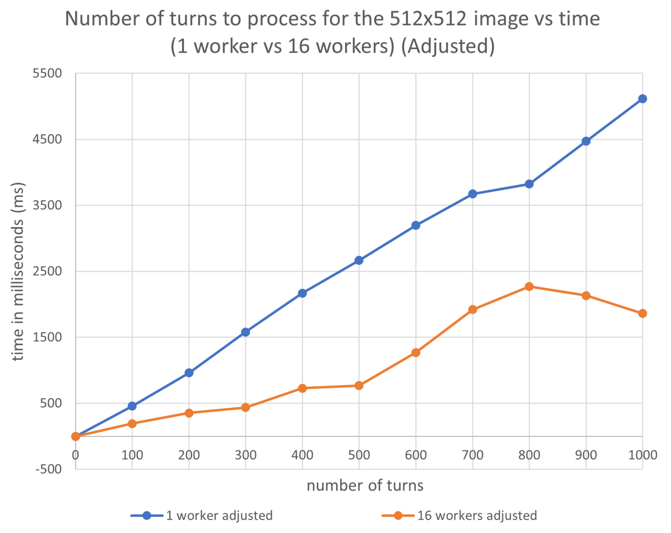

Fig. 6: Adjusted benchmark results comparing the time for the GoL logic for 1vs16 workers.

In Fig. 6 the time taken for 0 turns has been subtracted from the results of both benchmarks. As a result, we can deduce that with 16 workers the GoL logic calculation is 2.4x faster for 100 turns and 2.75x faster for 1000 turns compared to when only 1 worker is used. As was expected, the difference in speed scales much more consistently with the number of turns.

The *Worker*s are identical to their counterparts in the parallel version. The *Distributor* on the other hand has been stripped of much of the functionality of the parallel version such as event handling. This has resulted in the distributor being purely a coordinator for the workers. It initialises them, pauses them when commanded and swaps the previous and next worlds after each turn.

2 Distributed Design

Our distributed solution splits the parallel version into a client-side controller and server-side engine. The controller is predominantly responsible for handling user input while the engine autonomously performs the simulation. All communication between client and server uses remote procedure calls (RPC).

2.2 Data Transfer

The entire world is transferred from the client to the server as a 2D slice on starting a simulation. From then on the world is permanently kept server-side within the distributor. The workers can continuously update the world with minimal client communication.

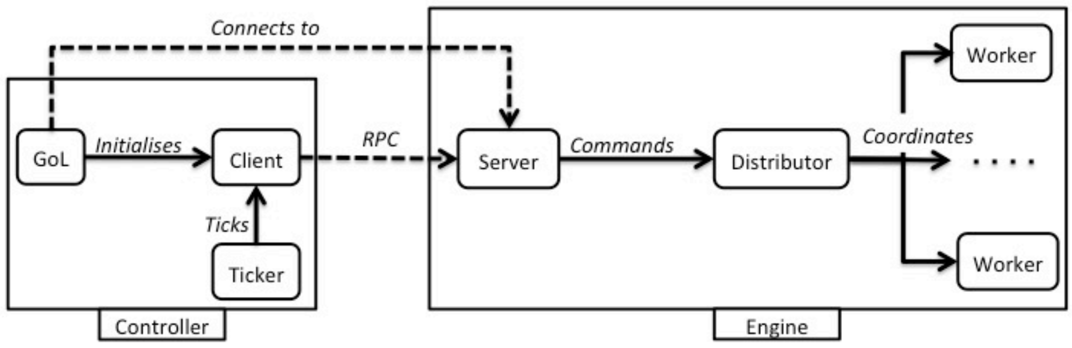

Fig. 7: Distributed architecture.

2.1 Components

GoL is used for initialising the IO goroutine and the client. It establishes an RPC connection with the server allowing the user to either start a new simulation or connect to an existing one.

The *Client* takes over all interactions with the server after the initial connection. It is responsible for handling user input and ticker events by making the appropriate RPC calls to the server.

The *Ticker* includes two tickers. The first ticks every two seconds prompting the client to get the number of live cells from the engine. The other ticks every hundredth of a second prompting the client to check if the simulation is done.

The *Server* is used exclusively as an interface for client communication. It includes the RPC functions that the client can call, each of which makes commands to the distributor. This decouples communication from the simulation itself, providing a modular structure that is easy to adapt during development.

Client-server interaction is strictly RPC only. This decision was made for design consistency, helping to simplify development. It is effective for almost all situations since all but one interaction can be modelled as a ‘question-response’ from client to server. The exception is on completion of the simulation where the server must send its final world to the client. By committing to using RPC for this, our client must constantly probe the server to determine whether or not it has finished. Although this solution is adequately fast, an improved version could make use of the publish-subscribe model. In this scenario the server would add each client to its subscriber list on connection and remove it if and when they choose to disconnect via the ‘q’ (quit) key press. This approach would allow the server to send the completed world to all subscribers when done, greatly reducing the number of calls made.

2.3 Additional Components

Our current solution could scale well by adding dedicated worker nodes to the system. Specifically, more AWS EC2 instances could be added to the system, each running a worker. The distributor would maintain a list of connected workers and make RPC calls to them. This would be done for synchronisation such as pausing and termination as well as transferring worlds when necessary.

Each worker would hold its own previous and next worlds that correspond to their virtual sections in the parallel implementation. In order to share top and bottom rows with their neighbours, each worker would keep the address of the neighbour above and below for a peer-to-peer model of communication. It could then actively send a copy of its own top and bottom rows to its neighbours after each iteration.

The effect of this approach is likely to depend on the size of the file being processed. For small files such as the 16x16 image used in tests, the overhead in communication between client, server, distributor and workers is likely to counteract any benefits. However for significantly larger images the use of multiple machines—each of which could themselves be parallelised—may allow for faster processing than could be possible on a single machine.

2.4 System Failure Considerations

We made sure to implement and test the controller's key press functionality so as to safely pause, disconnect the client and terminate the simulation. As well as this, we put a strong emphasis during development on decoupling communication from simulation processing. This allowed the server-side distributor and workers to autonomously process the simulation without further client interaction. If the client has problems and disconnects, the server is fully capable of continuing its task to completion.

On the contrary, the client must make RPC calls to the server. If the server goes down due to a fault or network issue, the client will receive an error on any attempted connection/call. If a fault develops on the server there is a risk that all progress on the simulation will be lost. A possible safety feature could be to have the server regularly save its progress while running by storing intermediate results to a file. This system of backups could enable the server to restore a previous simulation after a fault.

2.5 Performance Testing

In order to test the performance of our distributed solution we created benchmarks that would run on the client. The benchmarks used were identical to the ones run in the parallel implementation. We analysed the results against the previous ones in order to compare a distributed solution with a local one.

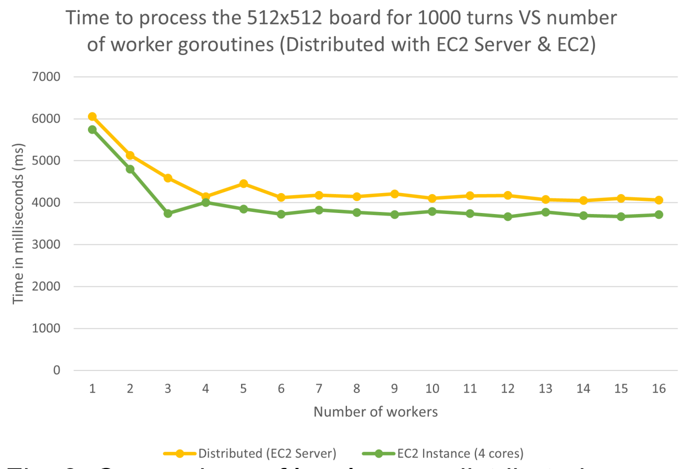

Fig. 8: Comparison of local run vs distributed run of testing worker numbers.

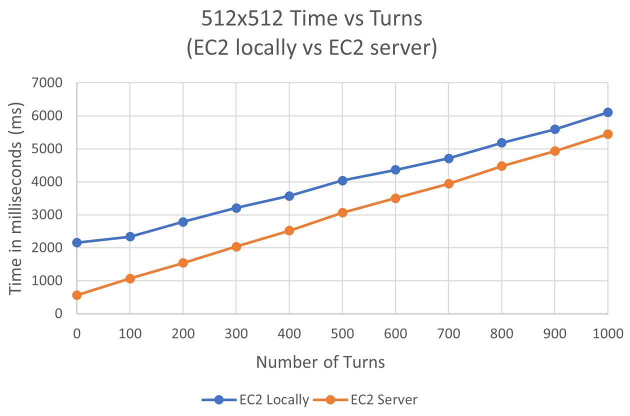

Fig. 9: Comparison of local run vs distributed run of testing number of turns.

The results of the distributed benchmarks follow the trend of the parallel ones since we have not managed to truly distribute the workload to many different machines. The GoL logic runs on a single computer just as before. Therefore, we are left with slightly worse times compared to running everything locally due to connection overhead, as can be seen in Fig. 8 and Fig. 9.

3 Conclusion

We believe we have successfully completed all of the steps in part one and all but one of the steps from part two of the requirements. Our parallelised approach minimises memory usage and splits the workload between threads. However the memory-sharing model we used cannot be easily adapted for distributed computing. This was one of the reasons we were unable to complete step three of part two within the timeframe. As for our distributed implementation, additional communication protocols could have been used to alleviate the issues caused by our exclusive use of RPC. Despite room for further optimisation, we think our code demonstrates good use of many of the concepts of concurrent and distributed computing that were taught in the module.