Concurrent Simulation of Conway's Game of Life

Lv20648 and Al20790

Stage 1 - Parallel Implementation:

Functionality and Design:

We initially implemented a single threaded simulation of Game of Life which involved reading an image containing the initial state of the board, transforming it into a 2D integer array and running the simulation for the specified number of turns. The image is read in and bytes are sent one by one down the IoInput channel which is received by the distributor and translated into a 2D array. The array is then looped over and by checking the state of the neighbours, the cell's new state is determined. After all the cells have been looped over, the turn is completed, and the board is updated with the new cells' states. After all the turns have been completed, the total number of alive cells is calculated and sent down the events channel in a FinalTurnComplete object.

After having completed a basic single threaded implementation, we started parallelizing our simulation. The board would be split into horizontal slices and for each thread, a worker would be assigned a part of the total board. For each worker, a channel was created to receive the processed slice after the Game of Life turn had been completed. Each worker was passed in the start and end indexes, the width of the image, the 2D array representing the world and the output channel. The workers run the same code as the serial implementation but send a slice, of a specified height, of the new state of the world down the channel. All of the channels are looped over, and the new world state is constructed by appending the slices to the 2D array in the correct order. The new world state is compared to the previous one and for each cell which has changed, a CellFlipped object is passed down the events channel to inform SDL to update and display the changes. There is a select statement which listens to keypresses from the user. If 's' is pressed, a go routine savePgm is called to convert the current world into an image and output it without pausing or disturbing the rest of the simulation. Pressing 'k' updates the loop counter to the number of turns specified in the Params, therefore breaking out of it, saving the final state as an image and terminating the program. If 'p' is pressed, processing of the current turn is paused until 'p' is pressed once more. This is achieved by creating a channel called pause and passing it to a go routine paused. This go routine listens to key presses and sends a signal down the pause channel when 'p' is pressed. A select statement causes the program to sleep until a value is received from the pause channel which then allows the program to continue processing the turn. This avoids the issue of busy waiting as the threads sleep instead of waiting and remaining idle, reducing CPU power consumption

Figure 1 is a diagram showing how the program works and interacts with all the goroutines

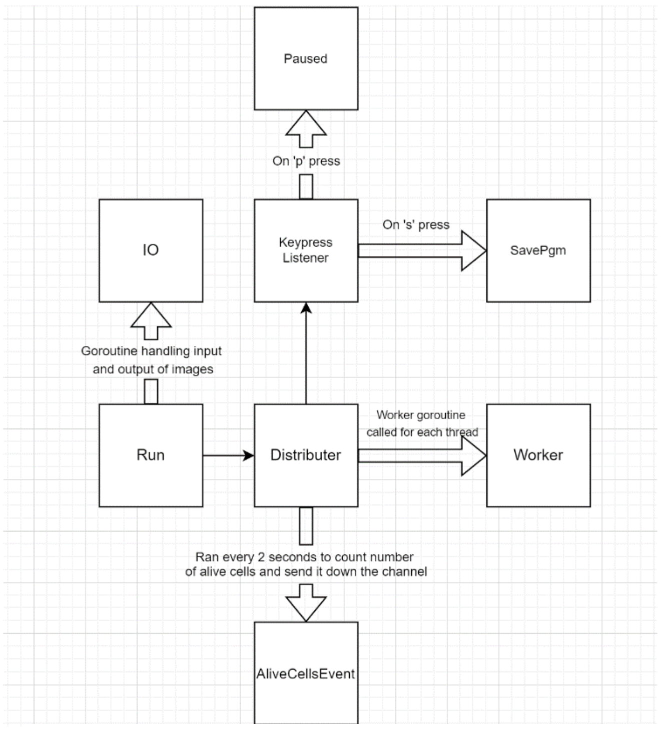

Paused
On 'p' press
IO
Keypress Listener
On 's' press
SavePgm
Goroutine handling input and output of images
Run
Distributer
Worker goroutine called for each thread
Worker
Ran every 2 seconds to count number of alive cells and send it down the channel
AliveCellsEvent

Figure 1

Problems encountered:

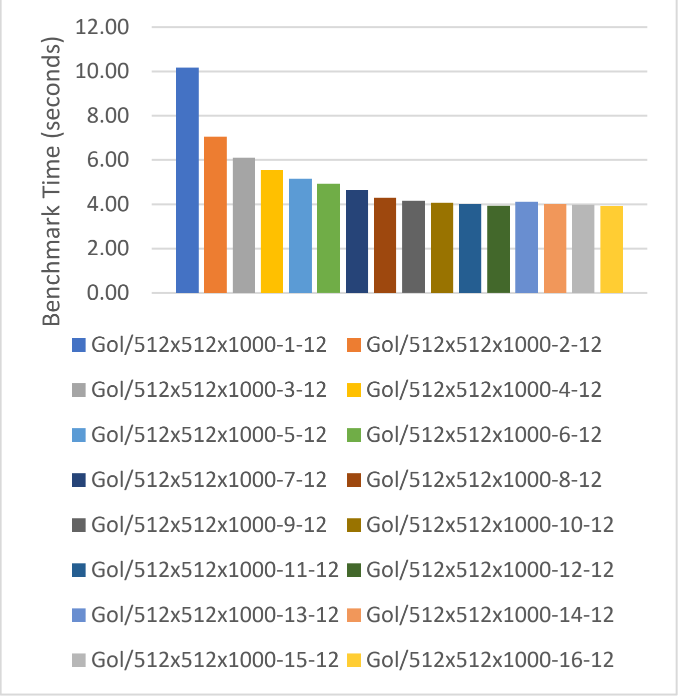
We initially implemented the worker function to receive a 2D slice of the world instead of the whole 2D array to improve efficiency. However, due to the nature of the board, cells on the edge of the board would need to know the state of the cells on the other side. This was only an issue if the cell was either at the top or bottom of the slice. This was rectified by passing the whole 2D array to each of the workers.

Another issue was how to deal with the cells on the edge of the board. We initially tried to write multiple if statements to handle when the program tried to access indexes out of bounds. However, we decided it was a more elegant solution to use the modulus operator to avoid this issue.

Critical Analysis:

To evaluate performance improvements due to parallelization, we used Go benchmarks. We varied the number of worker threads and measured the total time taken for the program to run 1000 turns of a 512x512 image. Each test repeated 5 times for each configuration to reduce the impact of random noise. The mean runtimes are presented in Figure 2

Figure 2



As the number of workers increased, the runtime decreased. The runtime plateaued at 10 threads with subsequent runtimes being only slightly faster than this. With 16 worker threads, the simulation was 2.61x faster than the serial implementation with 1 worker. The perfect improvement would be a runtime that halves as the number of threads is doubled. This is the perfect case scenario however it was not possible in this case because we also measured additional code ran and not just the Game of Life simulation. For example, the program needs to load the initial state of the board from an image and also output an image at the end of all the turns with the final state. While these events utilize channels, they are serial and cannot be parallelized. This means that regardless of the number of threads, there will be a constant time taken for part of the program which isn't affected by this.

The improvements also start to diminish, with the improvement from 8 to 16 workers being much smaller than from 1 to 2 workers. Code doesn't scale infinitely and since the machine I was testing on had a CPU has 6 cores, using any more than 6 worker threads offers virtually no improvement. The small improvement in runtime after this is probably down to hyperthreading with lets the CPU treat

Figure 3

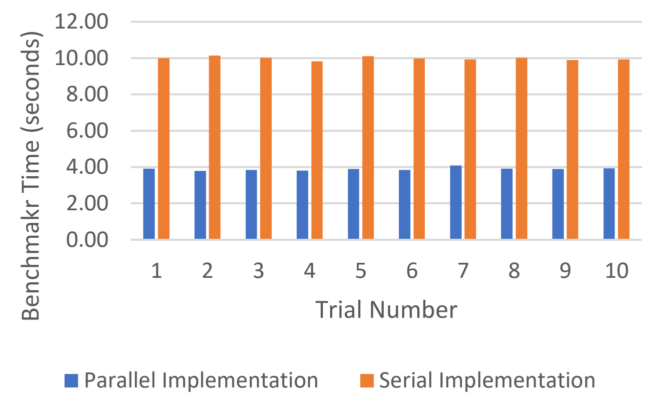


Figure 3 compares the benchmark times between the serial implementation and the parallel implementation with 16 threads. The benchmark was run 10 times with a 512x512 image for 1000 turns. This resulted in the parallel implementation performing 2.55x faster than the serial implementation.

Figure 4

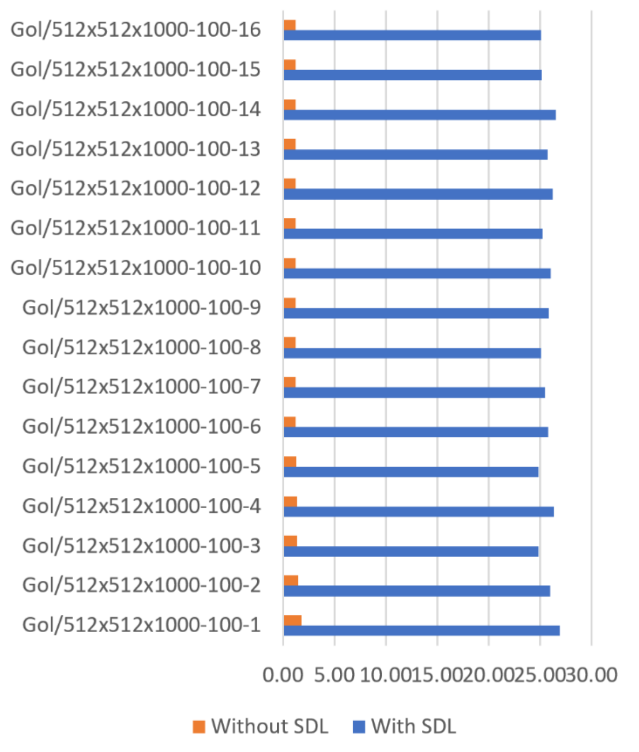


Figure 4 compares the benchmark speeds between running the simulation with and without SDL. This test was run 10 times with the 512x512 image for 100 turns. With SDL, the program ran with almost a constant runtime regardless of thread count. This is because displaying the graphic output of a 512x512 image is relatively CPU intensive and cannot be parallelized. On average, the benchmark run with SDL was 19.62x slower than without SDL.

Potential Improvements:

To speed up the runtime, we could replace the modulus operator as it is expensive. Instead, we could have a check which would make sure that the cell being accessed isn't out of bounds. 4 if statements would be needed, one for each side of the board which could be out of bounds.

Another improvement would be to have the board be a 2D array of Booleans instead of integers. This would slightly reduce the memory usage of the simulation.

Instead of using channels to send a slice of the new state, we could have used memory sharing to input the data straight into a new array. This array would be shared between all of the workers and all of them would be editing a different part of it so there would be no interference or overwriting of values. I think this would speed up our implementation slightly as it would avoid one step.

Conclusion:

Overall, our pure channel parallel implementation was significantly faster than the serial version. With 16 threads, it was 2.61x faster than with 1. While increasing the thread count above 8 led to diminishing improvements, this is probably due to the CPU only having 6 cores. Also, the all the slices from the channels had to be joined together to form the final world state.

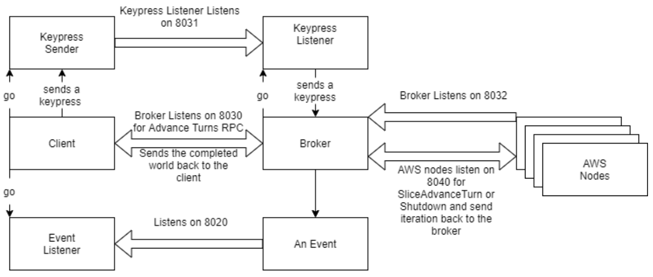
Stage 2 - Distributed Implementation:

Functionality and Design:

We started our initial implementation of the distributed by converting the initial step of the parallel implementation to being hosted using a client and server. A stubs file is used to set up the request and response structures as well as to list available methods for the client to call on the server.

The client could call the advance turns method of the server via stubs.AdvanceTurns and would pass in the initial world as well as how many turns to advance for. The result of the advancing the board by the number of turns specified is returned back to the client to be handled.

After this, the broker was the next obvious step forward with the implementation as it meant moving some more logic off of the client and onto the broker or server. The server is responsible for advancing a slice by a single turn. This allows for multiple servers to be working on different slices of the world, decreasing the runtime of program. To allocate servers to the broker, the broker will listen on port 8032 (Can easily be configured via command-line arguments to be other ports) for the required number of AWS nodes to send an initial message over to broker. This message is used to find the IP and port of the AWS nodes so that they can be used to generate the next state of the slice provided to them. Implementing the broker next made completing the additional tasks easier as it is mostly adding connections between the client and broker in the way as shown in figure 5.


Keypress Listener Listens on 8031
Keypress Sender
Keypress Listener
go
sends a keypress
go
sends a keypress
Broker Listens on 8032
Broker Listens on 8030 for Advance Turns RPC
Client
Sends the completed world back to the client
Broker
AWS Nodes
AWS nodes listen on 8040 for SliceAdvanceTurn or Shutdown and send iteration back to the broker
go
Listens on 8020
Event Listener
An Event

(Figure 5 – Diagram about network communication)

The order in which the 3 main parts are required to be started is as follows. Initially the broker needs to be started which will initially listen on port 8032 (can be configured via command line arguments) for all the specified AWS nodes (all ports can be changed and configured) to connect. Once the broker knows all the IPs and Ports of the nodes to RPC to, it will list on Port 8030 for RPC calls from the client. Now that the broker is ready, the client can be started up which will start up a listener on port 8020 which will listen for events sent from the broker. These events are "TURN", "ALIVE", "FLIPPED", "MESSAGE", and "SAVE".

TURN is sent along with the turn, after every turn so the client always knows what turn the broker and server are up to.

ALIVE is sent across every 2 seconds when the ticker fires. It also sends across the number of alive cells as well as the turn number.

FLIPPED, like turn is sent across every turn. It contains the list of coordinates that have changed state (have gone from dead to alive or alive to dead). This is required for the live view extension of the distributed.

SAVE is sent across when the broker decides that the current board needs saving to a pgm. Along with just the command, it will also send across the current board. This is done when the user presses S on the SDL window.

MESSAGE is sent across only around the pausing and is used to send the argument to the client to be printed. This could be used for debugging and future improvements.

As seen in figure 5, upon receiving an RPC call from the client, the broker will spawn a process listening on port 8031. When the client detects a keypress from the SDL window, it gets sent down, through channels, to the keypress sender which will send off a message to the broker containing the keypress. Upon arrival the keypress listener will send the key to the main broker thread to handle the input.

The SDL window for the distributed is a live view of how the simulation is progressing. This is achieved through utilising the FLIPPED events sent from the broker. This event was initially separate for each individual cell, but this severely affected the speed of the program, even when being hosted locally. To fix this problem, at the end of each turn, the flipped event sender is called and will build up a colon separated list of value of x and y that have changed.

This is all sent off in one single connection. The benefit of this is that now the code runs a lot faster as it more efficient to send a single longer piece of data than it is lots and lots of small messages. Another reason for this is that it will put less strain on the network as sending lots of messages on for example the 5120x5120 image would be equivalent to a denial-of-service attack and could cause the client's connection to drop, leading to the program crashing due to network problems. This is less of a problem if the client and broker connection is done via localhost, (with both being on the same machine) but may still be able to cause some problems.

Problems and Mitigation:

While developing the distributed aspect of Game of Life (GOL) I came across and had to solve many problems. The largest problem to solve was how to deal with needing multiple connections such as the broker listen on 3 different ports, send on 2 and then respond to the initial RPC.

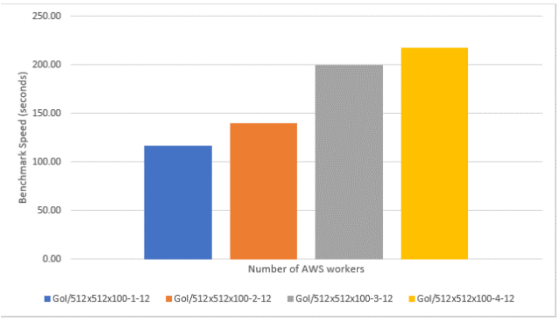
The fix for this issue turned out to be more trivial than expect as having go routines that are listening/sending on a new port whilst having a dedicated channel to communicate back and forth between them and the main broker routine.

Another huge problem we encountered was when testing the distributed with the AWS nodes. The way that the code is set up (see figure 5 for more detail) is that the broker will initially listen for the servers to connect. This was done to help with scalability in the future. However, it also means that the network the broker is sitting on has to be port forwarded to allow these connections from the AWS nodes to be actually sent to the broker.

Critical Analysis:

The development of the distributed was overseen using the tests provided running all on local host. All the tests that were provided passed.

After finishing the tests, we decided to benchmark the distributed using the AWS nodes.



(Figure 6 – Diagram showing time growth of more workers)

This data was gathered using t.2 micro instances on the 512x512 image for 100 turns as running for 1000 turns took quite a long time.

We were rather disappointed in our benchmarking results as they seem to suggest that by increasing the number of AWS nodes the time taken to run would increase (see figure 6). After reverting previous commits and trying to diagnose this problem we discovered that the initial test would crash in places due to "c.events" being closed , despite the fact that the close method had not been called on it.

Improvements:

There are a few improvements that could be made to the distributed part of the code.

The most obvious two improvements would be to fix the scaling issue presented in figure 6, and to still have all the tests passing in the current state. We were unable to determine if the scaling problem was due to the testing framework or was due to poor efficiency in the handling of the nodes. More testing around this would be needed and could be helped by comparing run times of the tests to see if this really was the case or not.

Another such improvement would be to allow for dynamically allocating workers between runs. This would be achieved by moving the listener for servers to a separate go routine within the broker. This would communicate with the broker via a channel. The broker could read from the channel after every turn and would allow for workers/servers to be added between turns (as long as the number workers requested is more than the current number of workers). You could also allow for servers/workers to be disconnected between turns by having this new go routine send all workers each turn rather than just the new ones. This wouldn't help with the problem that would occur if a worker were to disconnect whilst evaluating a turn. To fix this you would have to have a ticker that if timed out on a specific portion of the world would go to the next worker that was available to run that same section.

An extra improvement that could made is one that was mentioned earlier for the parallel part. This would be to replace the use of the modulus operator with checks that would allow for wrap around and would keep everything in bounds.

A final improvement that could be made would be to use halo exchange or another mechanism that would allow for less of the world to be sent to each worker. This would have multiple benefits such as decreasing the importance of the quality of the network between the broker and sever and having a lesser need of memory on the server. The way this works would be to have AWS node communicating with each other to share the extra data needed, other than just the slice the node is working on, between them. This would allow the turns to be calculated on the nodes themselves and would mean some logic would need moving across from the broker to the servers to handle this and the events.

Overall Conclusion:

The parallel segment of the project went exceptionally well with us managing to pass all the tests. Our critical analysis demonstrated that our utilisation of go routines was well implemented as the runtime would decrease as work was distributed across multiple threads. We even managed to show the differences between serial and parallelised versions of our code to show the speed increase from serial to parallel. On top of this we demonstrated how running the SDL window impacted performance.

Unfortunately, the distributed aspect of the project went less well, although there were quite a few positives that can be taken. The main negatives to this part were that the initial tests began to fail due to channels being closed after the final one had been completed, and the fact that there is a performance decrease as you increase the number of nodes, as suggested by figure 6.

There are positives to this side of the project than there are negatives. One such was the implementation of the SDL live view window which allows you to watch as the simulation takes place over the AWS nodes and the broker.

The second main advantage as slightly mentioned earlier, is how easily the program could be adapted to dynamically allowing servers to connect and disconnect between turns or between RPC calls from the client.

The last main positive is that the whole program runs correctly. The issues that are caused by the tests only occur after multiple calls of the RPC is done without the proper shutdown or channel closure.

Differences and Similarities:

The main difference between the parallel and distributed is the use of connections instead of channels. Channels are still used to communicate between go routines, but connections are mostly used in distributed as a lot of the required communication needed is between the client, broker, and servers. Connections are a lot slower than using just channels as they rely heavily on network speed and may need to be resent if packet loss occurs.

The main similarity between the two implementations is that most of the logic is shared between the two. This involves things such as how the work is split up between worker threads or AWS node and how key presses are dealt with. This heavily sped up the implementation of the distributed as logic and functions could be re-used or adapted to work with connections and RPC instead of channels.