Game of Life Coursework

Joshua Everett (td22885) Joshua Featherstone (qh22044)

I. Introduction

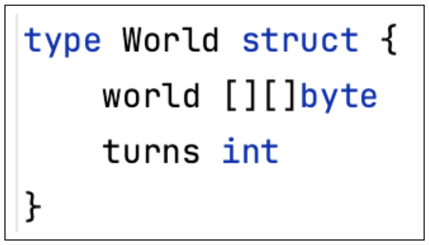
A. Summary

Our project involved implementing Sequential, Parallel, and Distributed implementations for the evolving state simulation Game of Life. The program is an interactive running simulation that evolves the initial 2D state given while communicating information back to output and taking input from the user to alter its execution.

II. Stage 1 - Parallel Implementation

A. Functionality and Design

For this section, we elected to use a primarily memory-sharing approach. We defined a struct World which contains both a 2D array and an integer. A pointer to this object is passed into our goroutines. It is written to in the main turns loop of the distributor and then read from in both the distributor and the separate goroutines.



```go
type World struct {
    world [][]byte
    turns int
}
```

Fig. 1. World Data Structure

Our Sequential implementation uses a single thread. This thread iterates over each position (i,j) and evolves that cell depending on the defined Game of Life Logic. The board is a closed domain so neighbors of cells at one extreme of the board are on the opposite extreme. Our Parallel implementation uses worker threads evolving sections of the world state separately.

B. Goroutines

KeyPresses listens for input from c.Keys channel. Save and Quit both safely read from World to output a PGM file then continue or os.Exit() respectively. Pause sends down the pauseTicker and pauseDistributor channel halting execution and triggering them to listen down that channel to restart. Ticker safely reads from World struct every 2 seconds and sends the Alive Cells Count event down the Events channel. It can be paused by keyPresses down the pauseTicker channel

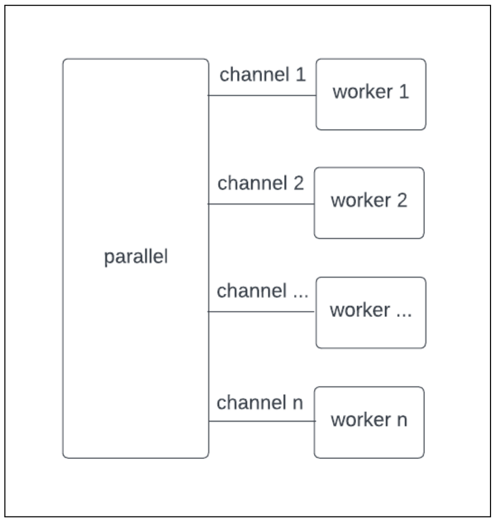
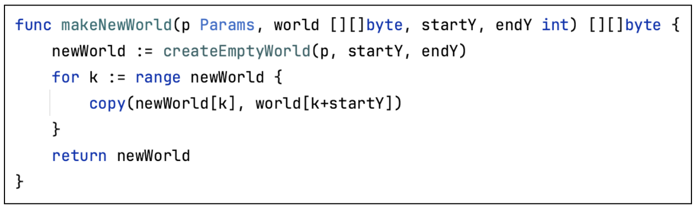


Fig. 2. Parallel and workers

Worker threads concurrently evolve their section of the world, using memory duplicates to prevent data races. Strips are calculated using multiples of p.ImageHeight / p.Threads and handed to a worker. Each strip is returned from the worker via its assigned channel and appended to a new world.

C. Problems Solved

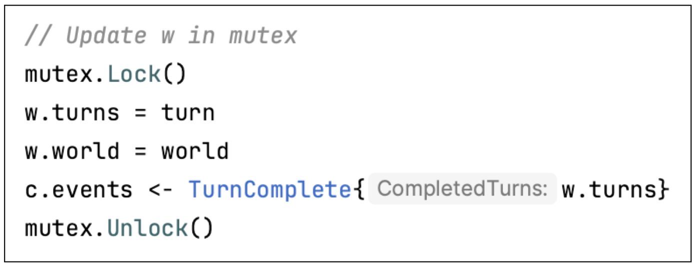
Our implementation uses an out-of-place algorithm with copies of the world array being made and passed around. This is sub-optimal for space complexity but ensures the correctness of our implementation. Correctness is crucial where any alive neighbors calculations must be done on the world state at the start of each turn and not on a partially updated one.



```go
func makeNewWorld(p Params, world [][]byte, startY, endY int) [][]byte {
    newWorld := createEmptyWorld(p, startY, endY)
    for k := range newWorld {
        copy(newWorld[k], world[k+startY])
    }
    return newWorld
}
```

Fig. 3. Out-of-place

We used mutexing for our World struct to further avoid any data races. A pointer to a Mutex Lock is passed into relevant goroutines to ensure only one thread can read/write to the World at once.



```
// Update w in mutex
mutex.Lock()
w.turns = turn
w.world = world
c.events <- TurnComplete{ CompletedTurns: w.turns}
mutex.Unlock()
```

Fig. 4. Mutexing World Struct

Within the c.Keys channel we encountered bugs where holding down a key would crash our program. We crossed this hurdle by clearing c.Keys after each computation of an s/q/p key in our goroutine.

D. Benchmarks

Using a benchmarking test we ran 10 iterations of our parallel implementation for each number of threads (1-16) allowing us to measure how the run-time changes with the number of threads. The resulting figure shows the average run time over those 10 iterations plotted. Comparing trends, the figure displayed a decrease in average run time as the number of threads increased.

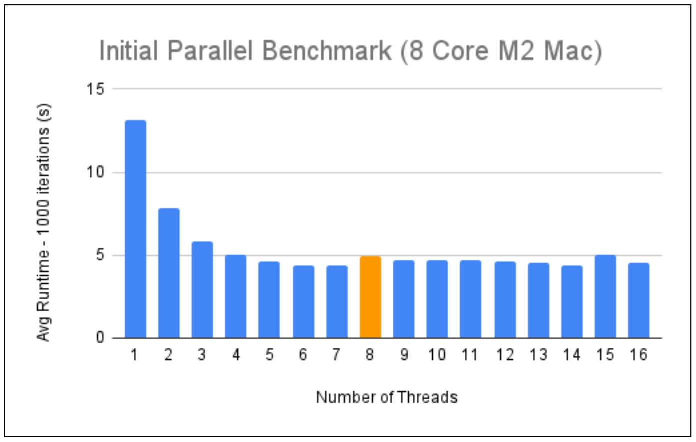
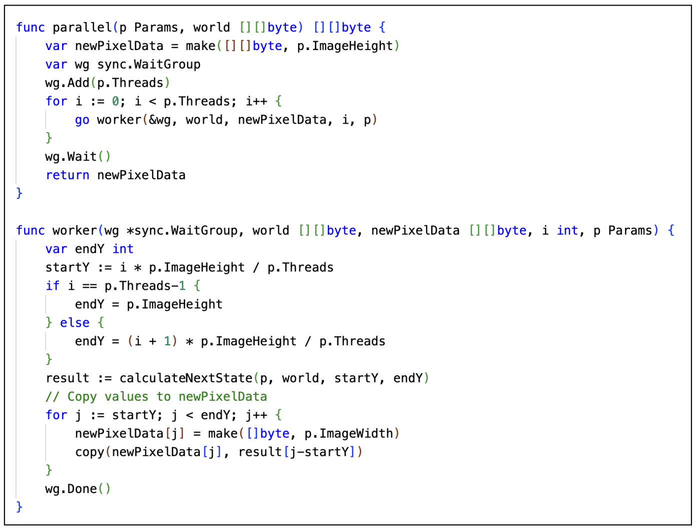


Fig. 5. Initial Benchmark

E. Data Analysis and Optimisation

In our initial benchmark the increase at the data point in orange implied there was an error in our implementation, creating an overhead that came with scaling number of nodes. We initially dived into improving our implementation by trying to better both space and time complexity. Slice Appending Overhead was removed by passing in an empty list to worker threads which is written into and returned from parallel. Wait Groups were implemented to improve the time complexity by returning our world exactly as workers finish. After these improvements had been added the overhead for our implementation was majorly improved creating a faster runtime output seen below.



```
func parallel(p Params, world [][]byte) [][]byte {
    var newPixelData = make([][]byte, p.ImageHeight)
    var wg sync.WaitGroup
    wg.Add(p.Threads)
    for i := 0; i < p.Threads; i++ {
        go worker(&wg, world, newPixelData, i, p)
    }
    wg.Wait()
    return newPixelData
}

func worker(wg *sync.WaitGroup, world [][]byte, newPixelData [][]byte, i int, p Params) {
    var endY int
    startY := i * p.ImageHeight / p.Threads
    if i == p.Threads-1 {
        endY = p.ImageHeight
    } else {
        endY = (i + 1) * p.ImageHeight / p.Threads
    }
    result := calculateNextState(p, world, startY, endY)
    // Copy values to newPixelData
    for j := startY; j < endY; j++ {
        newPixelData[j] = make([]byte, p.ImageWidth)
        copy(newPixelData[j], result[j-startY])
    }
    wg.Done()
}
```

Fig. 6. Parallel Improvements

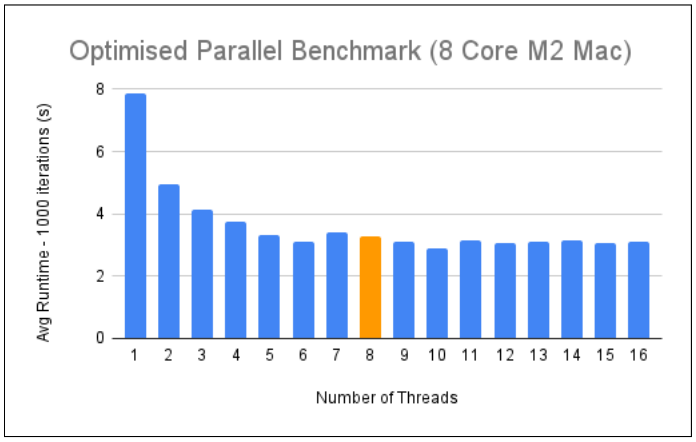
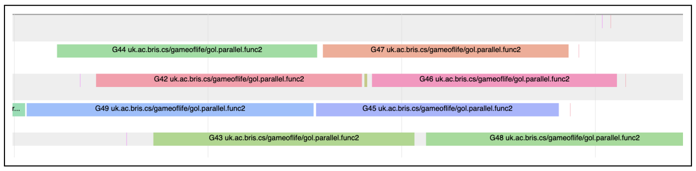


Fig. 7. Optimised Benchmark

This improved benchmark, despite being faster, still contained the anomaly we initially tried to remove where the improvements lessen around 8 threads and plateau. This led us to explore the tracing of the program to see if there was any reason for this.



G44 uk.ac.bris.cs/gameoflife/gol.parallel.func2
G47 uk.ac.bris.cs/gameoflife/gol.parallel.func2
G42 uk.ac.bris.cs/gameoflife/gol.parallel.func2
G46 uk.ac.bris.cs/gameoflife/gol.parallel.func2
r...
G49 uk.ac.bris.cs/gameoflife/gol.parallel.func2
G45 uk.ac.bris.cs/gameoflife/gol.parallel.func2
G43 uk.ac.bris.cs/gameoflife/gol.parallel.func2
G48 uk.ac.bris.cs/gameoflife/gol.parallel.func2

Fig. 8. Trace Section 1

Trace data received from one execution using 8 threads in two different sections is shown in figures 8 and 9. Each function in this section is an individual worker thread execution.

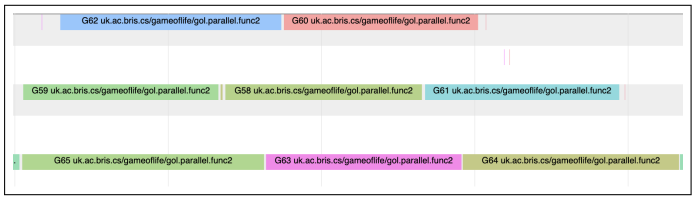

G62 uk.ac.bris.cs/gameoflife/gol.parallel.func2
G60 uk.ac.bris.cs/gameoflife/gol.parallel.func2
G59 uk.ac.bris.cs/gameoflife/gol.parallel.func2
G58 uk.ac.bris.cs/gameoflife/gol.parallel.func2
G61 uk.ac.bris.cs/gameoflife/gol.parallel.func2
G65 uk.ac.bris.cs/gameoflife/gol.parallel.func2
G63 uk.ac.bris.cs/gameoflife/gol.parallel.func2
G64 uk.ac.bris.cs/gameoflife/gol.parallel.func2

Fig. 9. Trace Section 2

Despite the goroutines being run concurrently in our code (using the go keyword), the Go run time scheduler appeared to be placing goroutines sequentially on a maximum of 4 threads. This meant that around 8 threaded executions the overhead from using more gorutines (both space and time) was greater than the minor improvement.

F. *Go Runtime*

Through the runtime scheduler in Go, goroutines are multiplexed onto the OS threads automatically. Go uses what is called an M: N scheduler which means it maps M green threads on N OS threads. The parallelisation is limited to these N threads as these are the number of available processors/cores. This means that in our case where OS threads are taken by say the startIO and gol.run gorutines dynamically the space for these wroker threads often requires them to be multiplexed sequentially. The scheduler also uses the concept of scheduling granularity where it may run goroutines in the same logical sequence to make use of shared resources (e.g. memory). This could have a contribution to our problem where the world state at each point is shared among all worker threads. Finally, to help balance the workload across processors if a thread is currently idle it employs Work Stealing. This is where it steals the work from another thread to help balance processing. We can observe this in action in Figures 8 and 9 where the work is split evenly across the multiple threads.

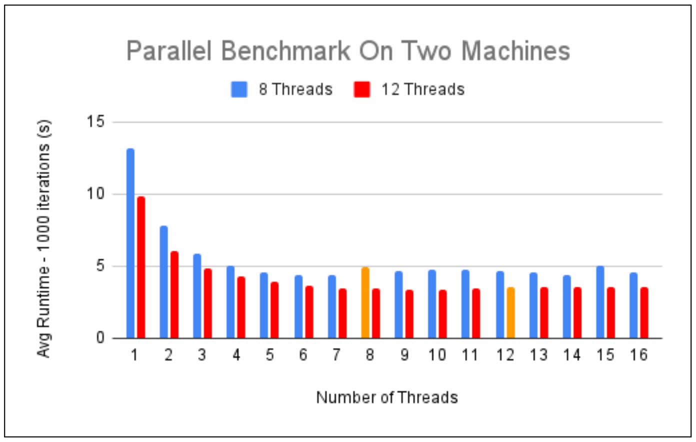


Fig. 10. Enter Caption

To confirm our suspicions we ran the initial benchmark on both the 12-core lab machines and 8-core M2 Mac and observed the same behavior with a peak around the number of cores.

III. Stage 2 - Distributed Implementation

A. *Functionality and Design*

For this section, we have built our design upon the structure below, with an emphasis on the distribution of processing across remote nodes. Remote procedure calls (RPCs) are used to send requests. The locally run distributor holds overall control, while a broker is called to delegate processing between external nodes. Stubs acts as a network interface, presenting available functions for machines. As per the diagram, there are three categories of machines communicating over the network. They are the local machine, the broker, and the nodes. Our stubs file gives a brief summary of the requests and responses that our RPCs use, so we have included a description for each function with the respective request and response in the next section.

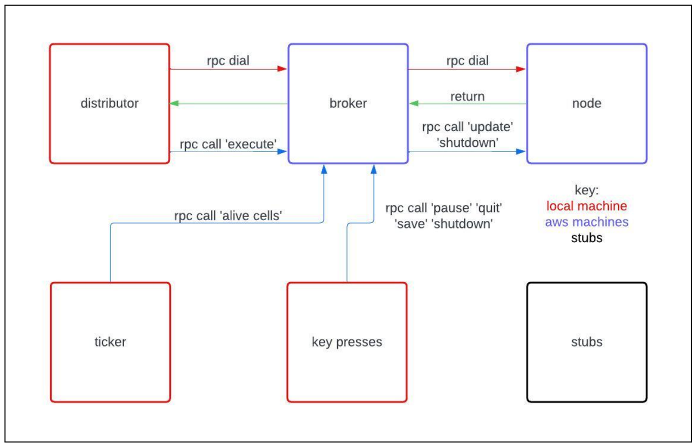


Fig. 11. Distributed System

B. *Network Communications*

Stubs contains the following handlers for each function in broker and node

```
var PauseHandler = "BrokerOperations.Pause"
var UnpauseHandler = "BrokerOperations.Unpause"
var QuitHandler = "BrokerOperations.Quit"
var ShutBrokerHandler = "BrokerOperations.Shutdown"
var SaveHandler = "BrokerOperations.Save"
var TickerHandler = "BrokerOperations.AliveCells"
var BrokerHandler = "BrokerOperations.Execute"
var GolHandler = "GolOperations.Update"
var ShutNodeHandler = "GolOperations.Shutdown"
```

Fig. 12. Stubs Handlers

The broker functions follow:

- Pause temporarily freezes broker processing. It is passed an empty *PauseRequest*, but the *PauseResponse* holds an int for turn number to be returned in.
- Unpause continues the broker processing after being frozen. It uses the same request and response structure as pause, however no turn count is returned.

- Quit is for ending calculations and returning turn count. It uses a basic *QuitRequest* and *QuitResponse* with the turn count returned as an int.
- Shutdown sets a local variable which will begin a safe shutdown process at the soonest opportune time. It uses a *ShutdownRequest* and *ShutdownResponse*
- Save returns the world state and turn count. *SaveRequest* is empty and *SaveResponse* holds the 2D slice and int for storing the current world state and turn count.
- AliveCells returns the alive cells and turn count. An empty *AliveRequest* and an *AliveResponse* with ints for holding the alive and turn counts are used.
- Execute oversees the updating of the world state. Here, *DistributorRequest* and *BrokerResponse* are used. The former holds game parameters and a 2D slice holding the current world state. This will be used to calculate the next state. The latter holds a 2D slice to return the new world via.

The node functions follow also:

- Update carries out the gol calculations on a defined strip, given a startY and endY, returning the same strip, but with the new state values. *BrokerRequest* and *NodeResponse* are used. The request holds the necessary startY, endY, game parameters and world state. These are needed by the node to carry out the correct calculations on the correct data. The response is a 2D slice to return the new world.
- Shutdownsimply terminates the node processing. The same *ShutdownRequest* and *ShutdownResponse* are used as broker uses. This causes the node to terminate processing immediately.

C. *Problems Solved*

Mutexing was again used to supplement our distributed implementation. The critical section of code we recognised was the for loop within the broker. A global mutex variable along with others was defined. This aided both stopping data races from separate rpc calls as well as giving us a clean solution for the pause/unpause. In both these cases we either grab or release the mutex to halt and resume the main execution

```go
var (
    alive     int
    turns     int
    mutex     sync.Mutex
    World     [][]byte
    quit      bool
    shutDown  bool
    nodeAddrs []string
    nodes     []*rpc.Client
)
```

Fig. 13. Global Variables

Our nodes can be dynamically scaled to any number with command line input. The broker takes in and parses a string list of broker ports it can allow nodes to connect to.

```go
// Parse node addresses from command line
var stringList string
flag.StringVar(&stringList, "stringList", "", "Comma-separated list of strings")
flag.Parse()
nodeAddrs = strings.Split(stringList, ",")
fmt.Println("Node Addresses: " + stringList)
```

Fig. 14. Command line scaling

D. *Extension - Parallel Distributed*

Due to the nature of our parallel implementation, it becomes easily applicable to our distributed code. Within our state module, which is responsible for updating the state, we can change one line of code to flip between distributed and parallel-distributed execution.

```go
func Next(p Params, world [][]byte, update chan [][]byte, startY int, endY int) {
    // Sequential if 1 thread
    world = calculateNextState(p, world, startY, endY)
    update <- world
    return
}
```

Fig. 15. Line 3 replaced with parallel function equivalent

E. *Benchmarks and Methodology*

We ran our benchmarks using AWS instances. Each node process was run on an instance as well as the broker. We set up these AWS instances to be m5.large in order to maximise their processing capabilities.

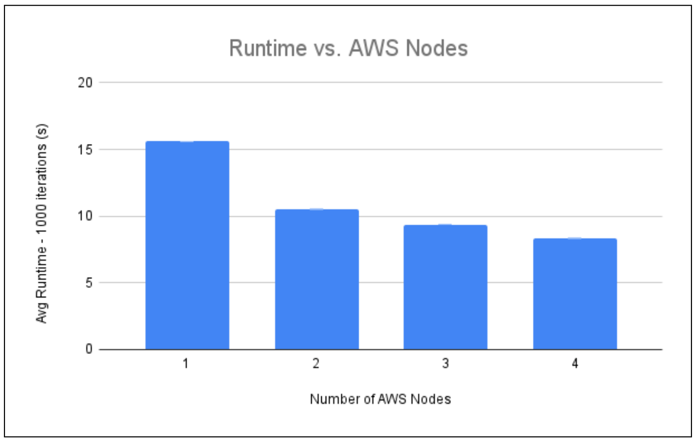


Fig. 16. Distributed Runtime Benchmark

As well as this benchmark we analysed our distributed parallel implementation running 4 AWS nodes each executing 8 threads in parallel.

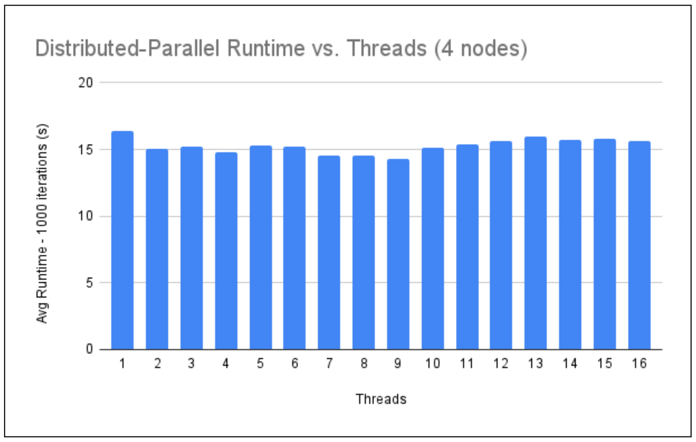


Fig. 17. Distributed Parallel

F. *Data Analysis and Optimisation*

Our distributed implementation showed the expected trend where increasing the number of nodes decreased the average run time. This is due to the processing of smaller sections across multiple processors being much faster despite the time needed to compile their separate outputs. These improvements resembled a similarity to the single-machine parallel improvements. In contrast, the distributed parallel implementation instead showed no marked increase in speed with threads. This was likely due to each AWS node not containing much processing it could benefit from in parallel, hence sequential execution was faster. Also in our parallel implementation, the gains began to deplete past the 8 threaded point. In the case of distributed parallel, we are splitting across 4 nodes of 8 threads (32 times) so much past this 8-thread threshold.

G. *Extension - SDL Live View*

Using two-way RPC we included dialling our Distributor from Broker. The RPC function in distributor SDL receives the cells that flip and turns passed via the SDLRequest struct. Given this information, it sends the Cell Flipped Events and Turn Complete Event down the c.Events channel. This allows the simulation to be visualised but adds a performance decrease as processing is taken up on both the local machine and AWS broker

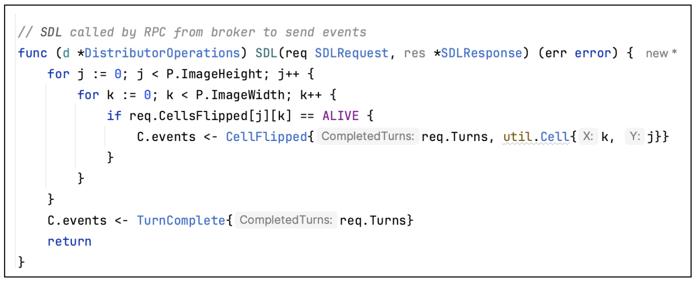

```
// SDL called by RPC from broker to send events
func (d *DistributorOperations) SDL(req SDLRequest, res *SDLResponse) (err error) {  new *
    for j := 0; j < P.ImageHeight; j++ {
        for k := 0; k < P.ImageWidth; k++ {
            if req.CellsFlipped[j][k] == ALIVE {
                C.events <- CellFlipped{ CompletedTurns: req.Turns, util.Cell{ X: k,  Y: j}}
            }
        }
    }
    C.events <- TurnComplete{ CompletedTurns: req.Turns}
    return
}
```

Fig. 18. SDL Live View

IV. Amdahl's Law

Amdahl's Law gives the theoretical speedup in execution when resources are improved.

$$S_{\text{latency}}(s) = \frac{1}{(1-p)+\frac{p}{s}}$$

where

- S_{latency} is the theoretical speedup of the execution of the whole task;
- s is the speedup of the part of the task that benefits from improved system resources;
- p is the proportion of execution time that the part benefiting from improved resources originally occupied.

Fig. 19. Amdahl's Equation

Applying this theory to our context, the section of code that benefits from parallelisation is the next state calculation. However, the setup, recompiling parallel states, and turn iteration are very much sequential. This law further improves our finding of diminishing returns as parallel/distributed components increase. The conclusion from this theory is that the more code is designed to be parallel the more returns you get from paralleising it.

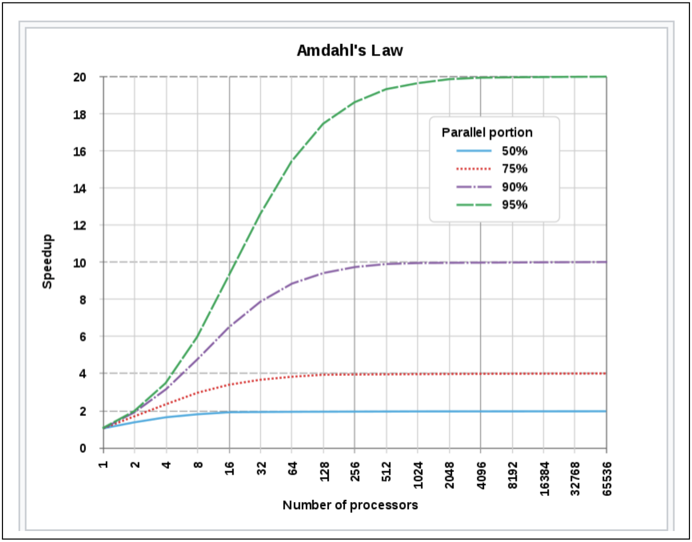


Fig. 20. Ahmdal's Graph

V. Conclusion

In conclusion, adding more threads in parallel and nodes in distributed improved performance of our algorithm. These gains became marginal as the number of threads and nodes increased hence obeying Amdahl's Law.

VI. References

Information about Go Runtime was taken from Concurrency in Go Cox-Buday, K. (2017). Concurrency in Go. 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media

Information about Amdahl's Law was taken from it's Wikipeida article 'Amdahl's Law'