**A concurrent multi-threaded implementation of Conway’s Game of Life**

*William Evans (qv18211), William Hannigan (yz18341)*

Team 45, University of Bristol, United Kingdom

3 December 2019

# Abstract

*A comparison of a concurrent implementation of Conway’s Game of Life over a single-threaded implementation. In this report, we aim to utilise goroutines to create a concurrent, multi-threaded Golang program simulating Conway’s Game of Life on an image matrix to achieve a faster implementation over a single-threaded program. We use CPU profiling to analyse the bottlenecks within different implementations of this simulation. Experimental results show that our concurrent implementation produces a speed decrease of 313.5% over our single-threaded implementation.*

# Introduction

Concurrent, parallel computation can allow programs to be made more efficient by splitting and distributing work so that it can be processed simultaneously. Traditional multi-threaded programs rely on the use of system threads to allow for concurrent, parallel computation. These traditional threads have problems due to the fact that data is communicate between threads through the use of shared memory. Utilising shared memory requires the use of features such as semaphores and mutual-exclusion locks (Lee, 2006) which lead to a slow down of processing. This implementation has been written in Golang. Golang is a concurrent, garbage-collected programming language (Golang.org, n.d.).

By writing this implementation in Golang, we are able to use Communicating sequential processes (*CSP*) style programming aiming to achieve more efficient concurrent, parallel computation. Through the use of goroutines and channels instead of traditional threads Golang.org. (n.d.) we utilise CSP style programming to reduce the amount of time our program spends locking and unlocking shared memory. Goroutines operate by being moved around between a set amount of system threads by the scheduler when one thread is blocked by a blocking operation in a goroutine. This method allows for concurrent applications to be built in a functional manner. By creating a concurrent, parallel implementation we aim to provide a performance increase over our single-threaded implementation by reducing the time it takes to process each turn of the Game of Life on an image.

# Functionality and Design

Our implementation first reads in the initial Game of Life board from a PGM image and sends this to a main distributor goroutine. The distributor goroutine is responsible for splitting up the image into *n* parts and communicating it to *n* workers byte-by-byte over a buffered channel for each worker, with worker each running as a separate goroutine. These separate workers operate on their part of the image and then communicate the new version of the image back to the distributor (also byte-by-byte over a buffered channel for each worker) where it is then reconstructed at the end of each turn in the distributor.

The image (*w x h*), where *w* is the width *h* is the height, is split into *n* parts. Each part has equal width *pW = w*  and are of equal height, if *n* is a power of 2 where the height of a part is . If *n* is not a power of 2, all parts are of equal height *pH* except part *n* which has height: , allowing the image to processed on any *n* workers where *n* is a multiple of 2.

Alongside this, user control has been implemented through the use of another *control* goroutine. This goroutine receives keypresses using the termbox function PollEvent() (Reinke, 2019). Keypresses are then sent to a key channel. At the end of each turn, the distributor attempts to read from the key channel and if a keypress is present, reacts in the following manner. Upon the letter q being pressed, processing is stopped and a final image is outputted depicting the final turn that has been fully processed. This is done using the *writePgmImage* function provided in pgm.go in the skeleton code. The world is sent byte-by-byte to the PGM goroutine using a channel. Upon the letter p, processing is paused until p is pressed again. Upon the letter s, an image is outputted depicting the current turn in the same way as the letter q but the program is not stopped.

A *ticker* goroutine has also been implemented which receives commands on different channels. The distributor sends the number of alive cells at the end of each turn on a *numAliveCells* channel which is received by the ticker goroutine. The number of alive cells is printed every 2 seconds through the use of a ticker provided by Golang’s time package (Golang.org, n.d.). Upon receiving a *p* keypress, the printing is paused by sending *true* on the ticker’s pause channel. This is then resumed by sending *false* on the ticker’s pause channel when another *p* keypress is received.

Through the use of channels and sending the image byte-by-byte on each channel, memory sharing is never used and therefore this concurrent implementation is thread safe. The current stage of this implementation solves the problem of executing our program on multiple cores in comparison to the initial single-threaded implementation which only makes use of a single system core.

# Experiments

We demonstrate the performance of both our single-threaded implementation and our concurrent, multi-threaded implementation by executing both implementations on the University’s lab machines with the CPU profiler and benchmarking tools provided by Golang’s tools (Golang.org, n.d.). By using the Golang benchmarking tool and the benchmarks provided in the skeleton code, we were able to measure the time to execute 1000 turns for each implementation on a varying number of threads. This was all tested on the 128x128 image provided in the skeleton code.

**Table 1: Runtimes for 1000 turns of 128x128 when using 2, 4 and 8 threads**

| **Implementation** |  | **Runtime (seconds)** |  |
| --- | --- | --- | --- |
|  | **n=2** | **n=4** | **n=8** |
| Single threaded | 0.720 | 0.719 | 0.718 |
| Baseline | 0.735 | 0.528 | 0.397 |
| Stage 3 multithreaded | 2.98 | 4.67 | 8.28 |

Table 1 shows the results of our runtime experiment. Each implementation was executed three times on the 128x128 image. The average runtime of all these executions was taken as the result to ensure that a fair test was conducted.

**Table 2: CPU usage for 1000 turns of 128x128 when using 2, 4 and 8 threads**

| **Implementation** |  | **CPU usage (percentage)** |  |
| --- | --- | --- | --- |
|  | **n=2** | **n=4** | **n=8** |
| Single threaded | 104 | 104 | 104 |
| Baseline | 185 | 296 | 394 |
| Stage 3 multithreaded | 145 | 172 | 203 |

Table 2 shows the results of our CPU usage experiment. Each implementation was executed three times on the 128x128 image and the CPU usage was calculated using Linux’s built in time tool returning a percentage, these commands were provided in the skeleton code. The average CPU usage of all these executions was taken as the result to ensure that a fair test was conducted.

Using the CPU profiler we analysed which top 4 functions requiring the largest amount of CPU time within the single-threaded and Stage 3 multithreaded implementation of our program. This was conducted using the *make cpuprofile* command provided in the skeleton code which outputs a CPU profile. This was then analysed using Golang’s internal tool *go tool pprof cpu.prof* followed by the command *top*.

**Table 2: Top 4 functions ordered by most CPU usage to least**

| **Implementation** |  | **Functions** |  |  |
| --- | --- | --- | --- | --- |
|  | **1st** | **2nd** | **3rd** | **4th** |
| Single threaded | getNumLiveNeighbours | getNeighbourLifeValue | distributor | getNewLifeValue |
| Stage 3 multithreaded | lock | unlock | chanrecv | chansend |

All tests were conducted on the same lab machine, when only one user was logged in to ensure that differences shown are just due to differences in implementation and not environmental differences.

# Results

From Table 1, we see that for n=2 threads, our single-threaded implementation outperforms all other implementations including the baseline (by a small amount).

# References

[1] Lee, E. (2006). The Problem with Threads. Computer, 39(5), pp.33-42.

[2] Golang.org. (n.d.). The Go Programming Language Specification - The Go Programming Language. [online] Available at: https://golang.org/ref/spec#Introduction [Accessed 3 Dec. 2019].

[3] Golang.org. (n.d.). Why goroutines instead of threads? - Design FAQ - The Go Programming Language. [online] Available at: https://golang.org/doc/faq#csp [Accessed 3 Dec. 2019].

[4] Reinke, G. (2019). nsf/termbox-go. [online] GitHub. Available at: https://github.com/nsf/termbox-go [Accessed 3 Dec. 2019].

[5] Golang.org. (n.d.). time - The Go Programming Language. [online] Available at: https://golang.org/pkg/time/#NewTicker [Accessed 3 Dec. 2019].

[6] Golang.org. (n.d.). Diagnostics - The Go Programming Language. [online] Available at: https://golang.org/doc/diagnostics.html#profiling [Accessed 3 Dec. 2019].