**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. 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.

# Method

Our implementation first reads in the initial Game of Life setup from a PGM image and sends this to a main distributor goroutine. The distributor goroutine is responsible for splitting up the image and communicating it to several worker threads, each running as a separate goroutine. These separate goroutines operate on different parts of the image and then communicate the new version of the image back to the distributor where it is reconstructed.

# 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].