COMS20001 Concurrent Computing

Coursework 1: Game of Life

This assignment is the 1st summative coursework. It runs over approximately 5 weeks. The coursework is worth 25% of the unit mark. It is to be completed in your programming pairs. You must report any change to your pairing to the unit director *before* starting your assignment.

Meet regularly and make sure you manage your team well. Let us know about issues before they grow to affect your team's performance.

Submission

Every student is required to upload their full piece of work (including all Go files, Makefile and your PDF report) as a single ZIP file to Blackboard before **17:30 04/12/2019**.

PLEASE FOLLOW THE SUBMISSION GUIDELINES

Make sure you submit it early (not last minute!) to avoid upload problems. Each team member has to upload an identical copy of the zip file containing all of the team's work.

Assessment

You will be marked on your code, system design, experiments and report, and your understanding of it. Your team is assigned a presentation time slot published on the unit website. Presentation sessions will be held over the last 3 weeks of term in MVB 2.11. Both team members must be present to get a mark. During the presentation, we will run and discuss the submitted program. We will ask you questions about your work and you will be able to showcase the merits of your project.

Do not plagiarise. Both team members should understand all code developed in detail.

Task Overview

Introduction

The British mathematician John Horton Conway devised a cellular automaton named 'The Game of Life'. The game resides on a 2-valued 2D matrix, i.e. a binary image, where the cells can either be 'alive' (pixel value 255 - white) or 'dead' (pixel value 0 - black). The game evolution is determined by its initial state and requires no further input. Every cell interacts with its eight neighbour pixels: cells that are horizontally, vertically, or diagonally adjacent. At each matrix update in time the following transitions may occur to create the next evolution of the domain:

- any live cell with fewer than two live neighbours dies
- any live cell with two or three live neighbours is unaffected
- any live cell with more than three live neighbours dies
- any dead cell with exactly three live neighbours becomes alive

Consider the image to be on a closed domain (pixels on the top row are connected to pixels at the bottom row, pixels on the right are connected to pixels on the left and vice versa). A user can only interact with the Game of Life by creating an initial configuration and observing how it evolves. Note that evolving such complex, deterministic systems is an important application of scientific computing, often making use of parallel architectures and concurrent programs running on large computing farms.

Your task is to design and implement a concurrent, multi-threaded Go program which simulates the Game of Life on an image matrix. The game matrix should be initialised from a PGM image and the user should be able to export the game matrix as PGM files. Your solution should make efficient and effective use of the available parallel hardware of modern CPUs by implementing farming or geometric parallelism with the help of goroutines and channel-based message passing.

Skeleton Code

To help you along, you are given a simple project, which showcases some of the functions and goroutines needed for the task. It reads in a PGM image and each turn flips white (alive) cells to black (dead) cells and vice versa. The skeleton is made up of:

- main.go - Where channels and goroutines are initialised.
- gol.go - Where the main processing of the Game of Life takes place.
- pgm.go - Where PGM images are written and read.
- control.go - Where a library called 'termbox' is used to intercept all keypresses during execution.

All functions in the above files have been thoroughly commented. If you have any doubts about any part of the skeleton please ask a TA for help.

Stages

Please note that partial marks will be awarded for all attempted stages as well as the understanding of concurrency you demonstrate during the viva examinations. Please do not give up if, for example, you are stuck on 1b: Still run your comparisons against the baseline, discuss results and exploit them as a vehicle for discussing concurrent principles.

Stage 1a (Up to 30%) - Serial Code

Implement the Game of Life logic as it was described in the task introduction. This should be a single-threaded implementation that will serve as a starting point in subsequent stages. Your Game of Life should evolve for the number of turns specified in `golParams.turns`. After completing all the turns:

- Output the final state of the board as a PGM image. Include the size of the board and number of turns processed in the filename.
- Create a slice of cells that are alive and send it on the `alive` chan.

Success Criteria: Demonstrate the correctness of your code by a) passing all tests specified in `main_test.go` and b) visually inspecting the PGM files of the final state of the board.

Stage 1b (Up to 40%) - Divide and Conquer

The Game of Life program developed in Stage 1a is single-threaded. Parallelise your Game of Life so that it uses worker threads to calculate the new state of the board. You should implement a distributor that tasks different worker threads to operate on different parts of the image in parallel. The number of worker threads you should create is specified in the `golParams.threads` field.

*Note that this should be a basic implementation where you **reconstruct the world in the distributor** after every turn (Stage 4 suggests improving upon this). You **must not** use memory sharing. Sending slices via channels or using the closure-based approach from lab 2 counts as sharing memory. Your solution must not have any race conditions.*

Success Criteria: Demonstrate the different behaviour of the parallel implementation vs. the single-threaded one. Use benchmarks to measure the performance of your parallel program. Explain the results obtained.

Stage 2a (Up to 45%) - User Interaction

So far we ran the Game of Life through the testing and benchmarking framework. It is also possible to run it directly with the `make gol` command. You may notice that once started, you have no control over the execution.

Use the `getKeyboardCommand` function provided in `control.go` to implement the following control rules:

- If `s` is pressed, generate a PGM file with the current state of the board.
- If `q` is pressed, generate a PGM file with the current state of the board and then terminate the program. Your program should *not* continue to execute all turns set in `p.turns`
- If `p` is pressed, pause the processing and print the current turn that is being processed. If `p` is pressed again resume the processing and print `"Continuing"`. It is *not* necessary for `q` and `s` to work while the execution is pauesed.

Success Criteria: Demonstrate working keyboard control and correct PGM outputs at all times.

Stage 2b (Up to 50%) - Periodic Events

The lab sheets included the use of a timer. Now using a ticker thread, print the number of cells that are still alive every 2 seconds.

Success Criteria: Demonstrate the correct count of cells is printed. Moreover, show that printing stops when the execution is paused with the `p` key and resumes when execution is resumed.

Stage 3 (Up to 55%) - Division of Work

Right now your implementation should allow for the number of worker threads to be equal to some **power** of 2 (i.e 2,4,8,16...). Make changes to your implementation so that it supports all **multiples** of 2 (i.e. 2,4,6,8,10,12...). Expand `main_test.go` so that the tests verify the 16x16 board on 6, 10 and 12 worker threads.

Success Criteria: Demonstrate the correctness of your code by passing the new tests. Discuss the challenges faced in dividing the work between your worker threads in your report.

Stage 4 (Up to 70%) - Cooperative Problem-Solving

Currently, you should be reconstructing the entire world in the distributor at the end of every turn.

Implement a halo exchange scheme where only the edges are communicated between the workers. The distributor should no longer be receiving the entire board from the workers after each turn.

Draw conclusions about the performance of your system in your report.

Success Criteria: Use the provided tools to analyse and compare your Stage 3 and Stage 4 implementations. Use Memory profiling, CPU profiling, a range of benchmarks as well as tracing. In your report, fully explain where the bottlenecks are in both implementations. Make sure that this version still passes all tests and satisfies the user input requirements from other stages.

Note that the implementation is worth 5% and the analysis is worth further 10%.

Stage 5 (Up to 80%) - Extensions

Option 1

Throughout this Coursework, you've used CSP-style concurrency - i.e. message passing. However, that is not the only way of writing concurrent code. Implement Game of Life using memory sharing. Write your implementation without using any channels. Analyse and compare your stage 4 and 5 implementations.

- Is it any faster than your Stage 4 implementation? Why / Why not?
- How do you prevent deadlock and race conditions?

Option 2

What if you were running your simulation on multiple CPUs? Implement some form of communication between 2 or more computers, for example with RPC, to allow further parallelisation. Analyse and compare your stage 4 and 5 implementations. *This hasn't been tested on the lab machines.*

- Is it any faster than your Stage 4 implementation? Why / Why not?
- How would your system react if one of your nodes went down?

Other options

There are many other possible extensions to Game of Life. If you'd like to implement something that isn't an option above you're welcome to do so, but please speak to the unit director first.

The missing 20%

20 % of the marks has been reserved for excellence **beyond the success criteria** described above. This can be demonstrated in both your written report and the viva.

Note that achieving these marks does not rely on attempting Stage 5.

Keep in mind that together with the marking indications outlined above, your understanding of your code during the lab presentation, as well as code readability will be considered for assessment.

Report

You need to submit a CONCISE (strictly max 4 pages) report which should contain the following sections:

Functionality and Design (1 page max): Outline what functionality you have implemented, which problems you have solved with your implementation and how your program is designed to solve the problems efficiently and effectively.

Tests, Experiments and Critical Analysis (3 pages max): Describe briefly the other experiments and analysis you carried out, provide a selection of appropriate results. Keep a history of your implementations and provide benchmark results from various stages. List the important factors responsible for virtues and limitations of your implementations.

Discuss the performance of your program with reference to the results obtained and indicate ways in which it might be improved. State clearly how fast your final system can evolve the Game of Life on all provided input images.

Make sure your team's names and user names appear on page 1 of the report.

Workload and Time Management

It is important to carefully manage your time for this assignment. You should not spend much over 25 hours on this task, this is about 5h per week over 5 weeks including labs. Do not spend hours trying to debug on your own; use pair programming, seek help from our teaching assistants during scheduled labs, use the forum, or see a lecturer.