ECE454 Computer Systems Programming
Homework 5: Parallelization

Michael Law - 997376343
Jonathan Ng - 997836141

1. Description of program

Neighbour-Count Cell Representation

With the given sequential_game_of_life implementation, each cell state is contained within a byte of information. With this underlying implementation, our first task is to make use of the space by only storing the state of the cell with 1 bit and using the rest of the bits to store extra information

Cell representation

Bit	7 6 5	4	3 2 1 0
information	unused	cell state	neighbour_count
description		1 for alive 0 for dead	represents the number of neighbours that alive this value can be zero to eight, in which we needs at least 4 bits to represent 8 (0b1000)

With neighbour_count calculated in load.c within our cell, we can avoid the continuous 9 reads of the neighbouring cells to obtain the neighbour count of the current cell by simply reading the neighbour_count in the current cell's first 4 bits. If the state of the cell needs to be changed, then we will update the neighbour_counts of the neighbouring cells. Our trade off is by reducing the number of reads to calculate the neighbour, we have replaced it with a number of writes to the neighbouring cells. However a cell state remains in the same state much more frequently than it does change state, providing us with almost a 3 to 4 times increase in performance with one thread.

Parallelization with pthreads

With the requirement of having shared memory in mind, each thread will have to wait for every other thread to finish before the next Game of Life (GoL) iteration. We accomplish this by having four threads, each taking a rectangular chunk of the GoL board, and the main thread waiting for all four threads to finish before the next iteration can occur. Four threads are used since there are four CPUs in the UG machines. Since each cell may access its neighbouring cells to incr/decr neighbour_count, we will process the first and last row of each chunk before parallelizing.

Compiler Flags Optimizations
We chose O3 because O3 represents the highest level of code optimization for the compiler, and by intuition the most optimized code runs the fastest. According to the GNU documentation, O3 compiles with a number of further optimizations from O2 including: -finline-functions, -funswitch-loops, -fpredictive-commoning, -fgcse-after-reload, -ftree-loop-vectorize, -ftree-slp-vectorize, -fvect-cost-model, -ftree-partial-pre and -fipa-cp-clone.

2. Performance Analysis

Wall Clock Timing (in seconds)

scenario / input file	**128.pbm**	**512.pbm**	**1k.pbm**
sequential_game_of_life	2.87	45.69	185.03
neighbour count (NC) implementation, 1 thread, compiled with O3	0.95	10.92	42.08
NC, 4 threads, compiled with O3	0.75	5.18	19.3

Speedup of Program (baseline / wall clock timing)

scenario / input file	**128.pbm**	**512.pbm**	**1k.pbm**
sequential_game_of_life	baseline	baseline	baseline
neighbour count (NC) implementation, 1 thread, traverse column-major, compiled with O3	3.02x	4.18x	4.40x
NC, 4 threads, traverse column-major, compiled with O3	3.83x	8.82x	9.59x

3. New Source Files

life_neighbour_counts.c

nc_game_of_life implements the described neighbour count implementation with four threads, and traverses in the column-major order for each iteration of GoL.
nc_game_life_life is returned by *game_of_life* in life.c