

# Milestone #1 Report

## Table of Contents

1. What My Project is About
2. What I Have Done So Far
  - a) Program Overview
  - b) Description of Main Functionality
  - c) Optimisation Steps
3. Performance
  - a) Steps Taken to Verify Correctness  
(Does it Work?)
  - b) Gauging Performance
4. References
5. Appendix
  - a) Data used for graphs
  - b) Sample profiling output
  - c) Command Line Options
  - d) Build Options

## ATTRIBUTION

The code in "gif.c" and "gif.h" was written by Charlie Tangora. It is not my code. I did not write it. My code supplies the data for the consecutive frames but it is the code in "gif.c" and "gif.h" that is responsible for encoding the data in the gif format and writing it out to a file.

# 1. What My Project is About

The aim of my project is to implement Conway's Game of Life [1].

In short, it is a simulation which is similar to replication in cellular biology. In practice it is a simulation which demonstrates that emergent behaviour can arise from incredibly simple rules.

The Game of Life is a zero-player game which consists of a two dimensional grid of cells. Each cell is in either an '*alive*' or '*dead*' state. During each '*tick*' (which is a turn of the game), the '*next*' state of each cell in the grid is determined by considering the cell's eight neighbouring cells (see figure 1) and applying the following rules:

- If the cell is alive and has 2 or 3 neighbours which are alive, it remains alive. Otherwise it dies.
- If the cell is dead and has exactly 3 neighbours which are alive, it comes to life. Otherwise it remains dead.

All of these evaluations occur simultaneously. The net effect of this 'simultaneous evaluation' is that during each tick the *next* generation is written to a different grid from the *current* generation<sup>1</sup>.

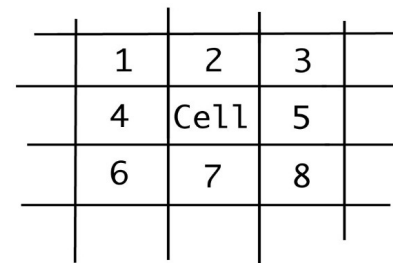


Figure 1: the eight neighbours of a cell in a Conway's Game of Life grid

As computer memory is limited, the dimensions of the grid must be finite. According to Wikipedia[1], "there are no pathological edge effects", meaning that Mr Conway did not specify what happens beyond the edges of the grid, thus it is left up to the implementation. As such, I have decided that everything outside the defined grid is dead. So any cell at the edge of the map is considered to have dead neighbours beyond that edge.

Another consideration is that most configurations of the initial grid quickly lead to either an empty state or a state where a cycle of a few grid layouts repeat indefinitely. This course has a focus on parallel computing, and a parallel implementation would soon arrive at one of these states and leave the program with very little to do. To alleviate this problem I have implemented an optional "random grid" mode (described in section 2a) which deviates from the rules described above. With this mode, there is a random chance during every tick that a *live* cell will die and that a *dead* cell will come to life. This will give the program something to do instead of repeatedly stepping from one empty state to the next.

---

<sup>1</sup> Incidentally, this alternating buffer idea is similar to the concept of double-buffering in graphics programming where the next frame of animation is written to an off-screen buffer and then 'flipped' into the graphics memory (which is what is displayed on the user's screen). This prevents the user seeing components of the frame being drawn.

## 2. What I Have Done So Far

### 2a) Program Overview

I consider my program to be of version 1 quality. It is feature complete, all input is validated, and I am confident that it behaves in the manner that I intend it to. It features:

- Command line parsing (see appendix C for a look at the command line options)
- Parsing of text input files which describe the initial grid layout. I have implemented two varieties of input file parsing.

"Map" files consist of repeated coordinates in the form of, "[10,10] [3,4] [3,8] [5,9]" where the first coordinate describes the dimensions of the grid and any subsequent coordinates describe the positions of the *alive* cells within that grid.

"Lex" files consist of an ASCII art representation of the Game of Life grid, such as those found at the [Life Lexicon](#), which represents *live* cells with a 'O' and *dead* cells with a period. (see figure 2)

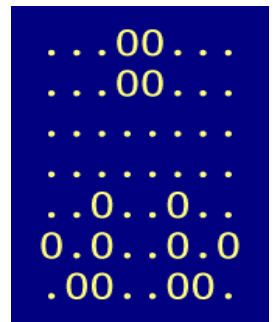


Figure 2: "glider-block cycle" An example of a "Lex" grid representation. [5]

Both parsers allow for "/" single line comments and generate errors if the input is invalid. There are command line options specifying the input file type ("-l" for lex files or "-m" for map files) but otherwise the program will *guess* the input file type based on the file's contents.

- The program features two main modes of operation; "default" and "random grid" mode. In default mode, the program will execute the game exactly as described in section 1 above, for the number of ticks specified on the command line.

In "random grid" mode, three parameters - optionally specified on the command line - are used to modify the 'basic' behaviour; initial life chance, life chance, and death chance. The *initial life chance* represents the chance that a cell is alive when the game begins. The *life chance* is a chance per tick that a cell which is dead will come to life and *death chance* is a chance per tick that a live cell will die.

- There are also some options for verifying the correctness of both the parsing and game running code. The command line options can be viewed in appendix C and the code will be discussed in detail in section 3a below.

## 2b) Description of Main Functionality

Conceptually, my implementation consists of two grids; the *current* grid and the *next* grid. The *current* grid represents the current state of the game. During each tick, the program visits each cells in the *current* grid, counts how many of its neighbours are alive, and applies the rules described above in section 1. In "random grid" mode, the cell now has a randomly determined chance of coming to life if it's dead, or dying if it's alive. The program then stores the cell's next *alive* or *dead* state at the cell's location in the *next* grid. Once all of the cells have been visited, the grids swap roles - the *next* grid becomes the *current* grid, and the (now old) *current* grid becomes the *next* grid. This process is repeated until all of the ticks have been completed.

Each of the grids is an array of width x height bytes, with each of those bytes representing one cell - either alive (1) or dead (0).

This is all done using the simplest code I could devise. It is designed to be easy to follow and to produce the correct results. Those results can then serve as a reference point for any optimisations which may be made, and for the later parallel versions. The optimisations which I have made are described below in section 2c. Code snippet 1 below is a simplified version of what the inner loops of the 'runLife' function look like:

(code snippet #1)

```
for(tick=0; tick < numTicks; tick++) { // number of ticks the game runs for
    for(y=0; y < gridHeight; y++) {
        for(x=0; x < gridWidth; x++) {
            aliveNeighbours = count_alive_neighbours(x,y);
            if (cell_is_alive(x,y)) {
                cellState = (aliveNeighbours == 2) ||
                            (aliveNeighbours == 3);
            } else {
                cellState = (aliveNeighbours == 3);
            }
            if (randGridMode) {
                r = random_double(); // 0 <= r < 1
                if (cellState) {
                    if (r < randDeathChance) cellState = 0;
                } else {
                    if (r = randLifeChance) cellState = 1;
                }
            }
            next_grid_store(x,y,cellState);
        }
    }
    curGrid = nextGrid;
    nextGridNum++;
    if (nextGridNum>1) nextGridNum = 0;
    nextGrid = grids[nextGridNum];
}
```

## 2c) Optimisation

### 2c.1)

The first optimisation came from wanting to write lazy code. The simplest implementation involves checking each cell's neighbours to see how many are alive (see code snippet 2). This means each check for each neighbour also involves checking whether that neighbour is outside the bounds of the grid. These "are this neighbour outside the grid" checks involve a lot of work (8 checks per cell) and they will almost always be false (they only ever be true if cell in question is at the edge of the grid) - and the larger the grid, the less proportion of that grid the edge cells will occupy.

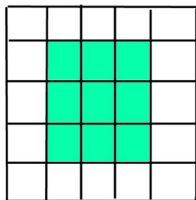


Figure 3: Example of a grid with a 3x3 active area expanded to 5x5.

So my first optimisation was to make the buffer for the grid two columns wider and two rows higher than the specified grid. So now the active area of the grid begins at 1,1 and ends at width, height. At the cost of a little more memory, the active area of the grid is surrounded by a line of empty cells.

For example if the program is to use a 3x3 grid, it instead allocates a 5x5 grid (as shown in figure 3) and uses the 1,1 to 3,3 area at the center of the grid. So when it comes to checking neighbours of the cell, it is safe to check the cells above, below, to the left and right of any cell in the active area without having to worry about reading past the end of the buffer or accessing a cell in at the other end of the grid by mistake.

The "count number of live neighbours" code goes from:

(code snippet #2)

```
numLive = ((x>0) && (y>0) && getCell(x-1,y-1))
          + ((y>0) && getCell(x,y-1))
          + ((x<Width-1) && (y>0) && getCell(x+1,y-1))
          + ((x>0) && getCell(x-1,y))
          + ((x<Width-1) && getCell(x+1,y))
          + ((x>0) && (y < height-1) && getCell(x-1,y+1))
          + ((y < height-1) && getCell(x,y+1))
          + ((x < gridWidth-1) && (y < gridHeight-1) && getCell(x+1,y+1));
```

to:

(code snippet #3)

```
numLive = getCell(x-1,y-1)
          + getCell(x,y-1)
          + getCell(x+1,y-1)
          + getCell(x-1,y)
          + getCell(x+1,y)
          + getCell(x-1,y+1)
          + getCell(x,y+1)
          + getCell(x+1,y+1);
```

Removing those dozen checks from every update of every cell is well worth the little bit of extra hassle of using 1-based indexing and using a little more memory to store the grids.

### 2c.2)

From checking the profiling output (see appendix 5b) 'getCell' (called 8 times per cell per tick) and 'setCell' (called once per cell per tick) are called so frequently, they are obvious targets for inlining. To

be honest, they were initially macros (which would guarantee inlining) but I made them into functions for easier verification<sup>2</sup>.

Admittedly, marking these functions as "inline" had no noticeable effect. Not counting the debug code, they are one line functions so I imagine they are already being inlined by the compiler, so my attempt to inline them did nothing.

### 2c.3)

Consider the grid in figure 4. When checking the neighbours of cell 'B', column c2 (cells: D,E,F) are the right-side neighbours of cell 'B'. This is not very useful when we're considering the next cell, 'E', but when checking the neighbours of cell 'H', the c2 cells are the left-side cells of 'H'.

Rather than adding up each column (c1, c2, c3, etc) twice, once when it's the right-side neighbour and once when it's the left-side neighbour of a cell, I thought it might be worthwhile caching the values of the previous two 'right-side' neighbours for when they become the 'left-side' neighbours of subsequent cells.

A	D	G	J	M
B	E	H	K	N
C	F	I	L	O
↑ c1	↑ c2	↑ c3	↑ c4	↑ c5

This observation provided a 12% performance improvement<sup>3</sup>. Though not earth-shattering, it's certainly not nothing.

Figure 4: Example grid

### 2c.4)

Another thing which I considered is to use more than two grid buffers. It turns out this has no effect other than making the program marginally slower because it has to allocate more memory.

### 2c.5)

Another option is to represent each cell with 1-bit in much the same fashion as I have done to represent the "correct" outputs which I discussed in section 3a. Essentially the entire grid can be represented with a bitmap. This would decrease the memory required for each grid to 1/8th of my current "one byte per cell" implementation. The cost would be to increase the amount of computation per cell with the bit-shifting and masking required. I considered it might be worthwhile to do in spite of the increased computation cost because the decreased memory usage would allow the grids to fit in a faster cache (eg. L1 cache rather than L2) and the faster memory accesses would overcome the slower computation speed.

Having now tested this idea, the '*bitmap*' version is slower for both pre-made maps and random maps. A characteristic comparison is for a pre-made 100x100 map, over 100,000 ticks, the '*char*' version of my program took 6.860 seconds while the '*bitmap*' version took 8.924 seconds.

<sup>2</sup> I added bounds-checking to these functions to ensure that the cell locations being accessed are at least within the grid.

<sup>3</sup> The mean time over 10 runs in "random grid" mode for 1,000 steps of 500x500 grid went from 7.3 to 5.7 seconds.

Unless there is a significant optimisation for the bitmap version which I have overlooked, the code is so much slower that even if the program could fit more data into a faster cache, the program still wouldn't work out faster overall. I have however left the 'bitmap' code in the source file in case I think of a way which would make this version faster than the 'char' (one byte per cell) version.

## 2c.6)

In "random grid" mode, there is a chance that a dead cell will come to life or a live cell will die during each tick. This chance is determined by the result of the runtime library function "rand()" converted to a double. I considered that converting an integer to a floating point value, then doing calculations on it is slow. I tried replacing the floating point code with similar integer code. The 'randLifeChance' and 'randDeathChance' parameters are converted to x-per-million integer values only once per program execution. This allows for easy and fast comparison with the "rand()" function's integer output. The result is that the code became roughly 18% faster<sup>4</sup>.

The code substitution was to replace:

```
(code snippet #4)
double r = ((double)(rand()*1000000)/1000000);
if (cellState) {
    if (r < randDeathChance) cellState = 0;
} else {
    if (r < randLifeChance) cellState = 1;
}
```

with

```
(code snippet #5)
int r = rand() % 1000000;
if (cellState) {
    if (r < randDeathChance) cellState = 0;
} else {
    if (r < randLifeChance) cellState = 1;
}
```

It is not a big difference but this code is executed once per cell per tick. At that scale, even tiny improvements add up!

## 2c.7)

The code to address a single cell looks like: `curGrid[(width+2)*y+x]`

During each tick of the 'runLife' function, the cells are processed from the top-left corner of the grid. The top row is processed from left to right, then the second row, etc. So rather than calculate the current cell's location every time, I thought it would be faster to have a 'current cell position' variable and use that to get the cell's state from the grid. eg. `curGrid[cellPos]`

After the current cell is updated, add one to 'cellPos' and it's referring to the correct offset for the next cell.

Also, rather than use the "`curGrid[(width+2)*y+x]`" to access each of the cells neighbours, it is a little faster to calculate the offset from the *current* cell for each of the neighbours. For example, the cell to

---

<sup>4</sup> The runtime decreased from 5.761 seconds to 4.713 seconds for a 500x500 grid over 1,000 ticks.

the left of the *current* cell is at 'cellPos - 1' and the cell below the *current* cell will always be at 'cellPos + gridWidth + 2'. So I surmise that calculating the offset of the cells neighbours prior to the inner loop will make the program a bit faster. ie.

curGrid[cellPos + ofsD] // ofsD is the offset for the 'down' neighbour

should be faster than

curGrid[cellPos + gridWidth + 2]

It turns out, this idea didn't pan out as I had expected. It actually made the program slower, though not by a huge amount. Comparing a "random grid", 1000 x 1000 size and 1000 ticks, the program went from 19.146 seconds to 19.258 seconds (these are the means of 10 runs). I surmise that this is a result of my 'optimisation' messing up what the compiler is doing to optimise the code.

## 2c.8)

One last consideration is the hashlife [4] implementation, originally by Bill Gosper. It works from the idea that the grid will quickly devolve into being sparsely populated, and the few areas which are still alive will have one of a handful of known patterns. So rather than process the whole grid for each tick, keep track of the small areas with *alive* cells and process those, and use look-ups to speed up the processing of the known patterns.

The downside of hashlife is that it consumes a rather more memory than even my 'char' implementation and, I suspect, would not handle my program's "random grid" mode - where cells can randomly come to life or die - at all well because hashlife benefits from being able to *remember* which sections of the grid have alive cells.



### 3. Performance

#### 3a) Steps Taken to Verify Correctness (Does It Work?)

- Any cell is only effected by its eight neighbours. This creates a 3x3 grid of 9 cells (see figure 1) of alive/dead values, which can be represented by 9 bits, which allows for  $2^9$  or 512 permutations. For each of those permutations, there is a correct *next* state. So my program has the option<sup>5</sup> to generate a look-up table consisting of the correct *next* state for each permutation. It also has a command line option ("-t") to test the current build - to verify that the current implementation of the 'runLife' function produces grids in accordance with that look-up table (see figure 5). During testing it also prints out both the input and output table for each permutation so one can visually verify that results are correct.

This look-up table can be used to verify that future iterations of this code will continue to produce the same, correct results.

- My program also has the option to generate an animated gif file where each frame describes one tick of the game<sup>6</sup>. This allows one to visually compare the output against other Game of Life implementations. For example, I have taken some of the initial patterns from the Life Lexicon [2] and compared the output of my implementation against Golly [3], which is a well known Game of Life implementation. While this 'blink test' approach is not exactly scientifically rigorous, it does at least give some confidence that my implementation is not producing wildly different output from other implementations. (NOTE: The code which generates the gif file, as found in "gif.c" and "gif.h" were not written by me. The extent my my code's involvement is to give it the frame data. It does all of the gif encoding and file writing involved in creating the gif file.)

- My program also supports two different input file formats (input files describe the initial grid layout). While the details are not terribly interesting, it does allow one to check one parsing routine against the other because one would expect the output to be identical regardless of which file format used to describe the initial grid. If the output is different then there is a problem.

- The program also has an option ("-p") which will load an input file in 'normal' mode or generate a grid in 'random grid' mode (see figure 6) and then print the grid to the standard output in 'lex' format. This allows visual confirmation that the file is being loaded correctly. It also allows one to load a 'lex' format file, the "-p" option

```

0.. 0..
000 ..0
000 0.0

test 506d (input: 01FAh expect: 0147h)
..0 000
000 ...
000 0.0

test 507d (input: 01FBh expect: 0145h)
00. 0.0
000 ...
000 0.0

test 508d (input: 01FCh expect: 014Ch)
..0 ..0
000 0..
000 0.0

test 509d (input: 01FDh expect: 0145h)
0.0 0.0
000 ...
000 0.0

test 510d (input: 01FEh expect: 0145h)
..0 0.0
000 ...
000 0.0

test 511d (input: 01FFh expect: 0145h)
000 0.0
000 ...
000 0.0

All tests passed.

```

Figure 5: Sample of output from a test run.

```

0 . . . . .
. . . . .
. . . 0 . . .
. . . 00 . . .
. . . 0 . . .
. . . . .
. . . . .
. . . . .
. . . . .
. . . . .
. . . . .
. . . . .
. . . . .
. . . . .

```

Figure 6: Example of a randomly generated 10 x 10 grid

<sup>5</sup> This is the "-c" command line option which can be viewed in appendix C

<sup>6</sup> This is the "-g giffle" command line option, also in appendix C

## Milestone #1 Report

will then print it to the standard output (which can be redirected to a file) which can then be compared to the original input file.

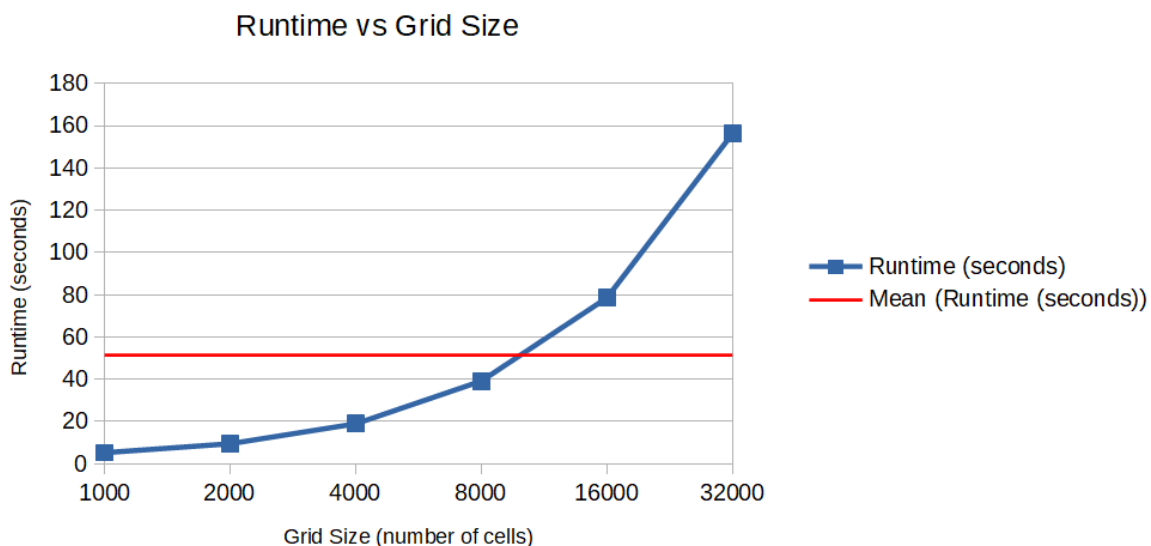
- By default the program uses the same random seed every time. This ensures it will produce the same output every time. For example, if you were to run  
`life -r -p 10 10`  
it will produce exactly the same *random* grid as seen in figure 6, and it will produce the same output every time. There is an option ("-a seed") to specify a different random seed value.

### 3b) Gauging Performance

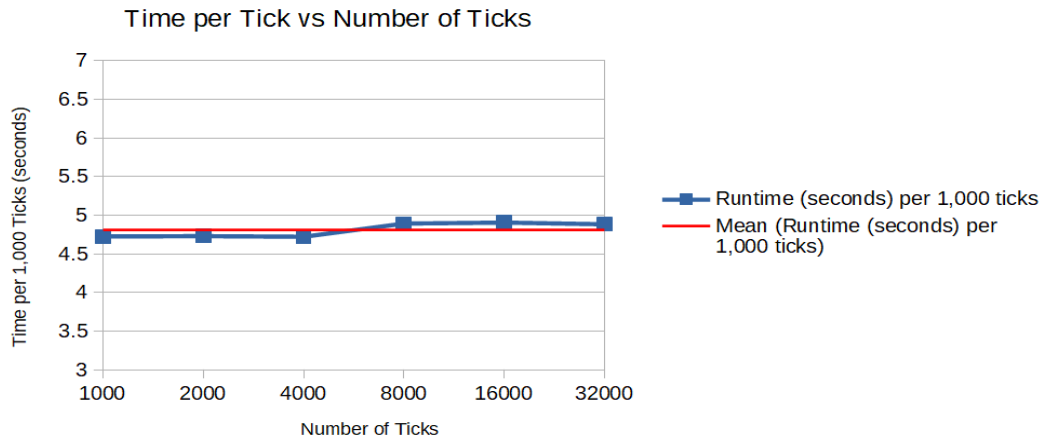
The first thing to mention is what is being measured. All of the values below are for:

- the 'runLife' function only. 'runLife' implements the program's inner loop. It is described in code snippet 1 above. Other sections of the program, notably the command-line parsing code, input file parsing code and verification routines are not considered.
- no output is produced during testing, either to a file or to the screen.
- the program is run in "random grid" mode using the default values for '*initialChanceToLive*' (chance a cell is alive in the first grid), '*randLifeChance*' (random chance a dead cell will come to life each tick) and '*randDeathChance*' (random chance a live cell will die each tick).
- the program build is the "release, char map, no floats build". ie. The release build is built without debug info using the -O3 compiler flag. This is the version of the program which uses a char to represent each cell (as opposed to the 'bitmap' version described in section 5.5). This version of the program has the floating point *chance* variables converted to n-per-million integer values (as described in section 2.6c above)
- The below measurements were taken on my laptop. (2 cores, Intel skylake i7 with 8Gb of RAM - in summary, a fairly recent machine but not a fast one)
- Times were measured using the 'measureTime' function, which in turn uses the 'steady\_clock' functionality from 'chrono.h'.
- Aside from the tests marked with an asterisk, the results are the mean runtime of 5 runs. Those marked with an asterisk were run once.
- The tables containing the data for the below graphs is in appendix A.

Runtime is obviously dependant upon the number of ticks the program runs for - the more ticks the longer it will take (see graph 1).

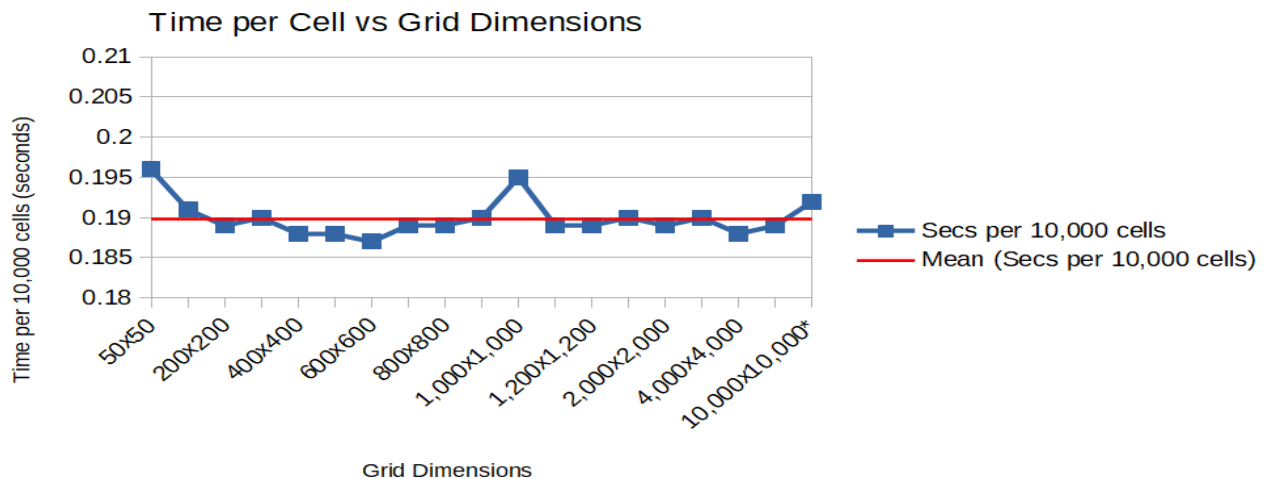


The time taken to complete a given number of ticks is independent of how many ticks are being performed (see graph 2). This shows a linear relationship between number of ticks and program runtime.



Graph #2

Scaling on number of cells is also linear. As the number of cells increases, the program quickly settles on a fixed time per 10,000 cells (see graph #3). I imagine that for the smaller grid sizes (50x50 and 100x100) caching data and code represents a substantial part of the program's runtime. This is why the time per cell values for these tests was higher.



Graph #3

On the whole, this program performs exactly as I would expect a serial implementation of the Game of Life to behave.

## **6. References**

[1] [https://en.wikipedia.org/wiki/Conway%27s\\_Game\\_of\\_Life](https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life)  
accessed on: 26-Aug-2020

[2] <https://conwaylife.com/ref/lexicon/lex.htm>  
accessed on: 26-Aug-2020

[3] <http://golly.sourceforge.net/>  
accessed on: 26-Aug-2020

[4] <https://en.wikipedia.org/wiki/Hashlife>  
accessed on: 26-Aug-2020

[5] [https://conwaylife.com/ref/lexicon/lex\\_g.htm](https://conwaylife.com/ref/lexicon/lex_g.htm)  
accessed on: 26-Aug-2020

## Appendix A

Table #1:

The below table show runtime for varying number of ticks with a 500x500 grid.

command line used:

```
life -r x 500 500
```

where 'x' is the number of ticks

Number of Ticks	Runtime (seconds)	Runtime (seconds) per 1,000 ticks
1,000	5.192	4.726
2,000	9.459	4.730
4,000	18.886	4.722
8,000	39.135	4.892
16,000	78.495	4.906
32,000	156.263	4.883

Table #2:

The table below shows runtime for 1,000 ticks at varying grid sizes. These are the mean of 10 runs except for the rows marked with an asterisk, which were only run once.

command line used:

```
life -r 1000 x x
```

where 'x' is both the height and width of the grid

Grid Dimensions	Number of Cells	Runtime (seconds)	Secs per 10,000 cells
50x50	2,500	0.049	0.196
100x100	10,000	0.191	0.191
200x200	40,000	0.756	0.189
300x300	90,000	1.707	0.190
400x400	160,000	3.014	0.188
500x500	250,000	4.710	0.188
600x600	360,000	6.775	0.187
700x700	490,000	9.281	0.189
800x800	640,000	12.101	0.189
900x900	810,000	15.393	0.190
1,000x1,000	1,000,000	19.550	0.195
1,100x1,100	1,210,000	22.890	0.189
1,200x1,200	1,440,000	27.193	0.189
1,500x1,500	2,250,000	42.703	0.190
2,000x2,000	4,000,000	75.953	0.189
3,000x3,000	9,000,000	171.152	0.190
4,000x4,000	16,000,000	300.848	0.188
5,000x5,000	25,000,000	473.006	0.189
10,000x10,000*	100,000,000	1926.915	0.192

## Appendix B

Sample gprof profiling of life.exe run in "random grid" mode, 1000 ticks, 500 x 500 grid

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
44.66	2.72	2.72	1	2.72	5.76	runLife_ch
40.39	5.18	2.46	150150000	0.00	0.00	getCell_ch
9.61	5.76	0.58	250015229	0.00	0.00	setCell_ch
5.09	6.08	0.31				getCellFromGrid_ch
0.16	6.08	0.01				rand
0.08	6.09	0.01	1	0.01	0.01	initGrids
0.00	6.09	0.00	3	0.00	0.00	std::chrono::duration<long long, std::ratio<1ll, 1000000000ll>
>::count() const						
0.00	6.09	0.00	3	0.00	0.00	parseInt64
0.00	6.09	0.00	2	0.00	0.00	std::chrono::time_point<std::chrono::_V2::steady_clock,
std::chrono::duration<long long, std::ratio<1ll, 1000000000ll>						>::time_since_epoch() const
0.00	6.09	0.00	1	0.00	0.00	parseCmdLine(int, char**, tagOutputOptions*)
0.00	6.09	0.00	1	0.00	0.00	parseCmdLineRand(int, char**, tagOutputOptions*)
0.00	6.09	0.00	1	0.00	0.00	std::chrono::duration<double, std::ratio<1ll, 1ll>
>::count() const						
0.00	6.09	0.00	1	0.00	0.00	
std::enable_if<std::chrono::_is_duration<std::chrono::duration<double, std::ratio<1ll, 1ll>						>::value,
std::chrono::duration<double, std::ratio<1ll, 1ll>						>::type std::chrono::duration_cast<std::chrono::duration<double,
std::ratio<1ll, 1ll>						>, long long, std::ratio<1ll, 1000000000ll>
>(std::chrono::duration<long long, std::ratio<1ll, 1000000000ll>						> const&)
0.00	6.09	0.00	1	0.00	0.00	std::chrono::duration<double, std::ratio<1ll, 1ll>
std::chrono::_duration_cast_impl<std::chrono::duration<double, std::ratio<1ll, 1ll>						>, std::ratio<1ll, 1000000000ll>
double, true, false>::__cast<long long, std::ratio<1ll, 1000000000ll>						>(std::chrono::duration<long long,
std::ratio<1ll, 1000000000ll>						> const&)
0.00	6.09	0.00	1	0.00	0.00	std::chrono::duration<double, std::ratio<1ll, 1ll>
>::duration<double, void>(double const&)						
0.00	6.09	0.00	1	0.00	0.00	std::chrono::duration<long long, std::ratio<1ll, 1000000000ll>
>::duration<long long, void>(long long const&)						
0.00	6.09	0.00	1	0.00	0.00	std::common_type<std::chrono::duration<long long,
std::ratio<1ll, 1000000000ll>						>, std::chrono::duration<long long, std::ratio<1ll, 1000000000ll>
>::type						
std::chrono::operator-<std::chrono::_V2::steady_clock, std::chrono::duration<long long, std::ratio<1ll, 1000000000ll>						>, std::chrono::duration<long long, std::ratio<1ll, 1000000000ll>
>, std::chrono::duration<long long, std::ratio<1ll, 1000000000ll>						>
>(std::chrono::time_point<std::chrono::_V2::steady_clock, std::chrono::duration<long long, std::ratio<1ll, 1000000000ll>						> > const&, std::chrono::time_point<std::chrono::_V2::steady_clock, std::chrono::duration<long long,
std::ratio<1ll, 1000000000ll>						> > const&)
0.00	6.09	0.00	1	0.00	0.00	std::common_type<std::chrono::duration<long long,
std::ratio<1ll, 1000000000ll>						>, std::chrono::duration<long long, std::ratio<1ll, 1000000000ll>
>::type						
std::chrono::operator-<long long, std::ratio<1ll, 1000000000ll>						>, long long, std::ratio<1ll, 1000000000ll>
>(std::chrono::duration<long long, std::ratio<1ll, 1000000000ll>						> const&, std::chrono::duration<long long,
std::ratio<1ll, 1000000000ll>						> const&)
0.00	6.09	0.00	1	0.00	5.76	measure_time
0.00	6.09	0.00	1	0.00	0.01	populateRandGrid
std::ratio<1ll, 1000000000ll>						> const&) [9] initGrids

(remainder removed for conciseness)

## Appendix C: Command Line Options

```
Life (release, char map, no floats build)
usage1: life [-v] [-s] [-w] [-l|-m] [d delay] [-g giffile] ticks infile

-d delay      Delay between gif frames, measured in MS
               (default: 50)
-g giffile     Name of gif file to produce
-l            Force lex parsing
-m            Force map parsing
-s            Silent operation.  No output unless error occurs
-v            Verbose output
-w            Display warnings
ticks         How many ticks to run the simulation for
infile        Name of the input file to act as the seed

usage2: life -r [-a seed] [-i init%] [-l life%] [-t death%]
             [-g giffile] [-d delay] [-v] [-s] ticks width height

-r            Generates a random map
-a seed       Seed for random number generator (default: 987654321)
-i init%      Initial chance for a cell to be alive (default: 0.020)
-l life%      Chance per tick for each dead cell to come alive (default: 0.001)
-t death%     Chance per tick for each live cell to die (default: 0.000)
-d delay      Delay between gif frames, measured in MS
               (default: 50)
-g giffile     Name of gif file to produce
-s            Silent operation.  No output unless error occurs
-v            Verbose output
-w            Display warnings
width         width of the grid
height        height of the grid
ticks         number of ticks to run the simulation for

-- Verification options --
life -t       Test generator code for verification
life -c       Generate 'correct' values for testing
life -p [-l|-m] input_file  Parses the input file, prints it out and quits
life -r -p [-a] [-i init%] width height
                               Generates a random grid, prints it out and quits
```



## Appendix D: Build Options

My program was build on my Windows laptop using TDM-GCC-64. It is a build of mingw (the windows port of GCC) which supports Open-MP, which I consider will be useful as this course proceeds.

The 'Makefile' has the following options:

**clean:** deletes OBJ and EXE files.

**debug:** produces a build with debug info for profiling.

**release:** produces the compiler optimised version without debug info.

**profile:** profiling build for use with gprof.

This makefile submitted with this project has been tested on Windows, Linux Mint 20, the UQ "moss" server and the hpc cluster.