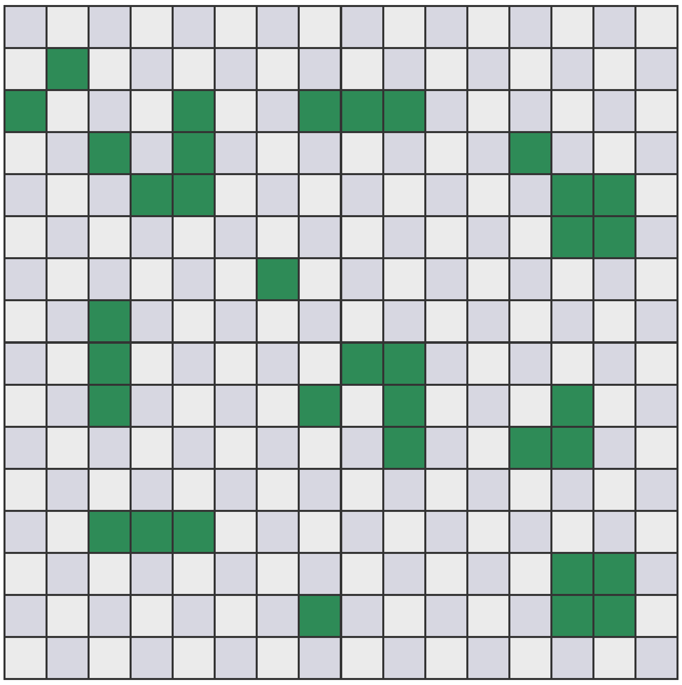
Parallelisation of the Game of Life

Assignment 1: Report

Dylan Carpenter [21982288]



The University of Western Australia
CITS3402: High Performance Computing
16th September, 2018

Preface

The implementation of The Game of Life analysed in this report uses the Von Neumann neighbourhood model of checking only the vertically and horizontally adjacent neighbors to each cell. The attached code was written in the c programming language with the OpenMP API to parallelise the code.

Note - compile source code with: *$ gcc -std=c99 -fopenmp -o gofl gofl.c*

All timing of execution times was done using OpenMP function calls returning a double-precision floating-point number, and as such provided times may not be accurate past the microsecond level. All averages were calculated over 100 trials of 100 generations unless otherwise stated.

Contents

1 Introduction

1.1 Aim

In the modern day, most computers possess multi-core processors which can be used to perform parallel computations to the result of vast improvements in the execution time of programs. With parallelisation, many mathematical simulations can run at greatly increased speeds, particularly when simulating extended periods of time.

Through the use of the OpenMP parallel computing API, parallel programming techniques have been used to increase the performance of simulating Conway's Game Of Life. Investigated features include:

- varying thread utilisation
- parallelisation of nested loops
- methods of work distribution

Experimentation with these techniques has produced insights into methods of optimising (or minimising) the performance of celular automata simulations.

1.2 The Game Of Life

The Game of Life is a cellular automaton created by mathematician John Conway in 1970. It consists of an infinite plane of cells which are either 'dead' or 'alive' and evolve over a numbers of generations. At the end of each generation, subsequent cell states are evaluated with the following rules:

1. Any living cell with less than 2 neighbors will die of "loneliness"
2. Any living cell with more than 3 neighbors will die of "overcrowding"
3. Any dead cell with exactly 3 neighbors experiences a "birth" and becomes alive

This implementation of the Game of Life uses the Von Neumann Neighborhood and hence only considers the four vertically or horizontally adjacent cells as neighbors. Additionally, to approximate an infinite plane, edges of the grid wrap around on themselves in a toroidal fashion. Figure 1 shows four successive generations using the Von Neumann neighborhood on a wrapping, 16×16 board.

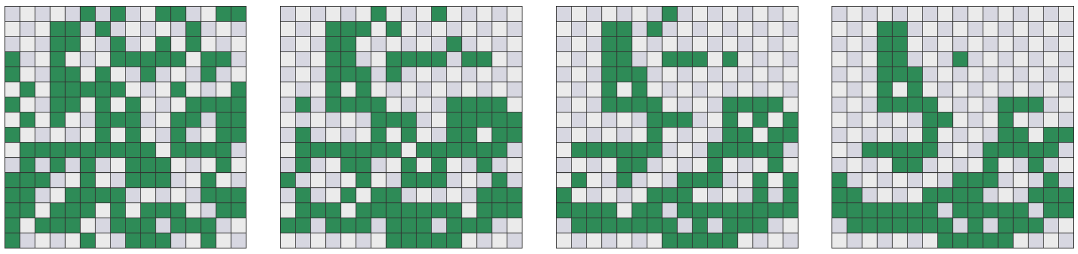


Figure 1: Four consecutive Game of Life generations (left to right)

2 Implementation Experiments

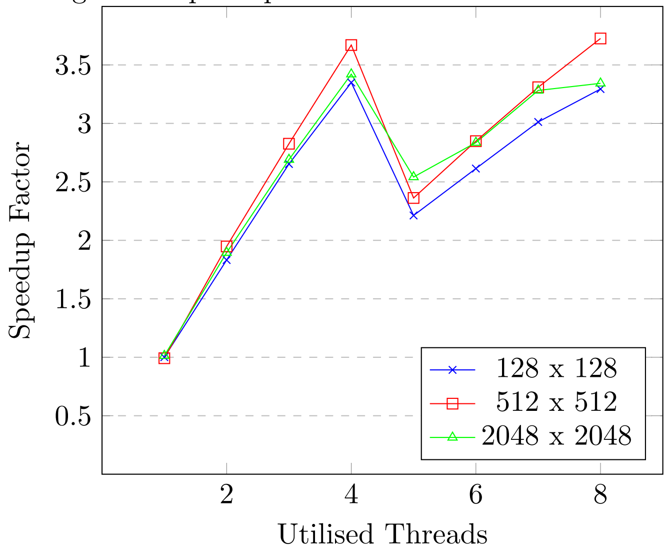
To optimize the effectiveness of multi-threading, several techniques were used to reduce the overall execution time of the simulation. These experiments are independent of the algorithm used to run the simulation, and only vary the way in which parallel computation is used to increase performance.

2.1 Thread Usage

For multi-threaded programs, varying the number of utilised threads has the most significant affect on execution time. For highly repetitive programs such as Game of Life, threads can concurrently execute loop iterations, theoretically allowing a high degree of speedup.

Code was analysed by timing exectution with varying numbers of threads over several input board sizes. These times were compared to the sequential execution times for the relevant input size. It should be noted that tests were run on a hyper-threaded CPU with four physical cores operating as eight logical cores.

Figure 2: Speedup Factor and Thread Count



As seen in figure 2, performance peaks when four threads are used, with a speedup factor of approximately 3.5, which is roughly matched by eight threads. This is quite atypical as highly iterative programs typically are able to approach linear speedup. The poor speedup is likely a result of cache thrashing and the use of a hyper-threaded CPU.

Cache thrashing describes the phenomena of slowdown caused by repeated cache-misses. As each thread is repeatedly required to access not only contiguous memory location (checking horizontal neighbors), but also from further away (checking vertical neighbors), the thread constantly needs to refill its cache due to the comparatively small cache line size. This is described further in the discussion section.

To compound the cache thrashing, the use of hyper-threading means that one or more logical cores are sharing a single L1 cache. This likely increases the frequency of a cache-miss, and hence increases cache thrashing. Despite this, having more threads still increases the performance of the simulation, this benefit is just obfuscated by the excessive cache-misses.

2.2 Parallel Loops

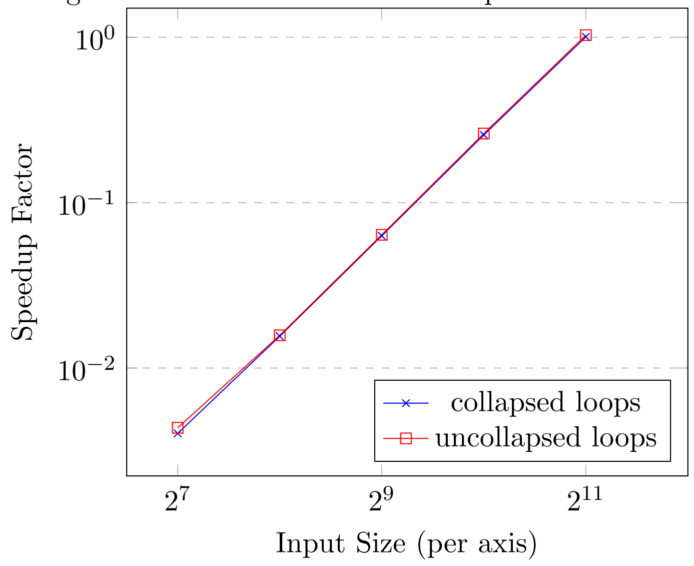
By default, OpenMP will only parallelise loops one level deep, which significantly decreases the time taken to iterate over a given block of code. This behavior can be extended to operate over nested loops by use of the OpenMP clause "`collapse`". Consider the code below:

```
...
//outer loop
for(int x = 0; x < n; x++){
    //inner loop
    for(int y = 0; y < n; y++){
        evaluateCell(x, y);
    }
}
...
```

If the code above was compiled with the directive "`#pragma omp parallel for`" before the outer for loop, threads would be made to execute full passes of the inner loop, causing each thread to call `evaluateCell()` a total of n times before being reassigned. When collapsing the loops and parallelising the inner `for` loop, "`collapse(2)`" is appended to the OpenMP directive and iterations of the inner loop are distibuted between threads. This result in threads being given a single call of `evaluateCell()` at a given time. This has no effect the overall amount work done by each thread.

To test the effect of parallelising loops, the Game of Life simulation was timed over a variety of board sizes, with ans without the parallelism of nested loops.

Figure 3: Execution Time and Loop Parallelism

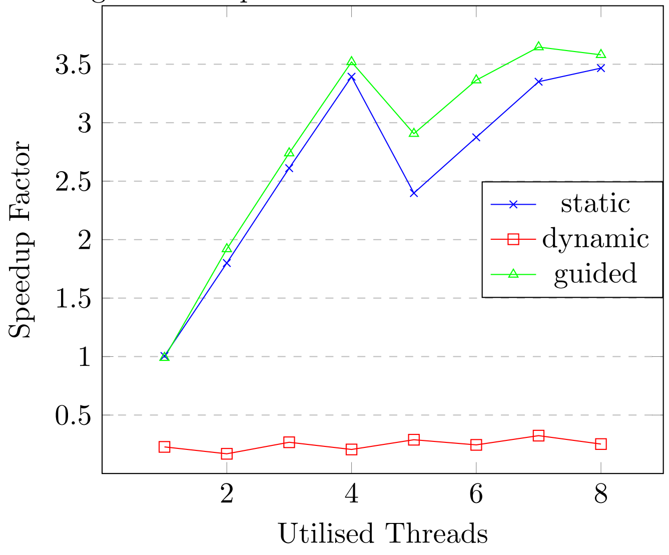


As seen in figure 3, this had a negligible effect on the overall execution time of the program, with the parallelised loops marginally faster on average over the tested input sizes. This likely is also a result of cache thrashing, caused by repeated swapping of the memory in the cache. As each thread will likely have to swap its cache three times per cell, there is almost no performance difference between threads processing individual cells or processing full rows.

2.3 Loop Chunk Allocation

In processing a `for` loop, threads must be assigned a specific subset of the loop's iterations to perform. This can be done in a variety of ways, each with different effects on performance. The effect of these different methods was tested by running the simulation of a 512×512 board over multiple threads counts. The results are as follows:

Figure 4: Loop Chunk Allocation Method



The default method of dividing work is `static`, which allocates work evenly across the threads and achieved comparable results to those in figure 2. The slowest method tested was the queue-utilising `dynamic` mode, which due considerable overhead performed consistently and considerably worse than sequential execution independent of thread utilisation. The fastest method, `guided`, functions very closely to `static` but attempts to balance the load between threads as execution continues; reducing the time spent idle at thread barriers. This mode had very little effect if less than four threads were utilised, however, as shown in Figure 4 resulted in significant performance gains when utilising logical cores by compensating for the effect of L1 cache thrashing between neighboring threads.

3 Discussion

3.1 Final Implementation

The final implementation Game of Life combines the most optimised solutions explored in the previous section. These include:

- utilising four or eight threads (depending on system default);
- parallelising nested loops; and
- distributing work with `guided` schedule mode.

This configuration however, does ***not*** generate an acceptable speedup factor, particularly on hyper-threaded systems. This is due to the several factors, predominantly cache thrashing, which is explored in the following section.

3.2 Impact of Cache Thrashing

Cache thrashing is caused by repeated cache-misses; the result of frequently requiring different segments of memory to be moved into cache. This problem is particularly prevalent in simulations of the Game of Life, as the calculation of each cell requires memory from distant locations (when checking vertically adjacent neighbors) as well as contiguous ones (horizontally adjacent neighbors). Because the size of cache lines are very small compared the the size of rows of data, multiple cache entries are required to evaluate the state of each individual cell. This results in the cache being repeatedly overwritten, and causes excessive idle time during which data is being fetched from memory and copied into cache.

In order to reduce cache thrashing, it is necessary to make major changes to the method of simulating generations. These methods could include array padding, using more advanced data structures, utilising hash tables or permanently storing additional data about cell neighbors. These methods will likely mitigate the effect of cache thrashing, but come at the cost storage. Additionally, solutions such as array padding are typically device independent, and implementations utilising this technique sacrifice both portability and large amounts of memory.

3.3 Other Influencing Factors

In addition to cache thrashing, there are several other factors which affect the performance on the simulation.

Due to the rules evaluated at each generation, the operation for determining the state of dead cells is significantly faster than for alive cells. This means that generations with higher densities of living cells will result in a larger execution time. Subsequently, due to the volatility of cellular automata, it is difficult to maintain a a consistent density without using input which already in in steady state. The inconsistent densities has high potential to damage the validity of tests conducted.

An additional problem is the use of a hyper-threaded CPU for testing. As the algorithm used to calculate cells is composed mostly of array accesses, which are made slower by cache thrashing, the effectiveness of logical cores is heavily reduced. Logical cores approximate physical cores by undertaking calculations in idle periods during a cache entry. As the algorithm has a very low proportion of other calculations, logical cores have few tasks to accomplish during idle periods and hence have a small or negative effective on overall performance.

Finally, another potential problem is the use of dynamically allocated memory. Much like cache thrashing, a lack of contiguous memory can cause unnecessary cache entries. Due to the large data sets processed by the simulation, memory must be allocated dynamically. This was accomplished with calls to the function `malloc()` from the C Standard Library, which does not guarantee allocated memory is in contiguous blocks. The extent of the effect of this, while most likely negligible, is unknown and may have contributed to the execution of the code.

4 Conclusion

Conway's Game of Life can have notably improved simulation times through the use of parallel computing. Techniques such as loop parallelisation, work allocation and thread utilisation can be used to progressively improve performance, however these improvements can be overshadowed by larger inefficiencies caused by low level functions of modern computer architectures.

The implementation of the Game of Life presented offers restricted multi-threaded performance and sub-optimal execution times, due to a high degree of cache thrashing, but presents a reasonably memory efficient solution to the problem. Improvements to this program would certainly sacrifice memory efficiency for time efficiency and would require a fundamental change of method to optimise the use of high-performance parallel computation techniques.