Game of Life using Parallel Programming Techniques – Profiling

**Introduction**

For this assignment, a form of Cellular Automata has been implemented. This is known as the Game of Life (GoL) by John Conway. A 2D array of elements are present, with cells being shown being filled in as alive or unfilled as dead. Each state in the Game of Life is iterated by a set of rules that each cell in the 2D array follows. These are;

* Any cell that’s alive and has fewer than two live nearby cells dies.
* Any cell that’s alive with two or three nearby live cells lives on to the next iteration.
* Any cell that’s alive with more than three live nearby cells dies.
* Any dead that’s dead with three live nearby cells becomes alive.

In the accompanying C++ code, two versions are presented. One is a sequential version using a nested for-loop to process the 2D arrays and the other is a parallel version using Intel’s Threaded Building Blocks (TBB) library. Both of these versions have extensive testing detailed below.

**Experiments**

In the following experiments, different situations of the GoL are present. Each are tested multiple times using both sequential and parallel versions with their averages being used as our final data sets. The results provided analyse the average time taken for each update loop, as well as total time taken to complete 600 cycles of our code.

These tests have also been carried out on multiple CPU’s, each with varying cores and threads available. The CPU’s used to test these parameters are;

Desktop - Intel Core i7 7700K @ 4.2 GHz (4.5 GHz Turbo) – 4 core CPU with 8 threads.

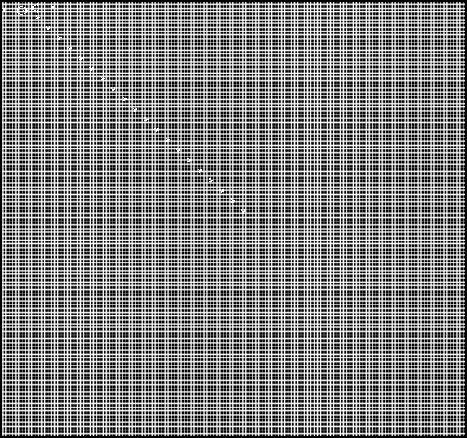
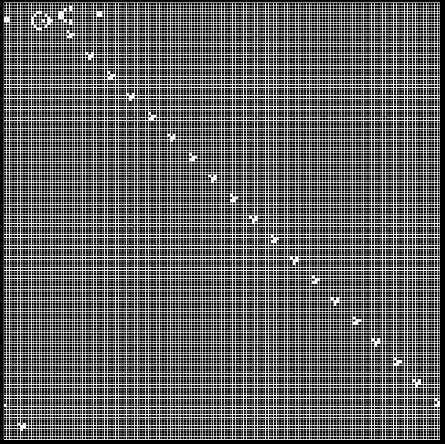
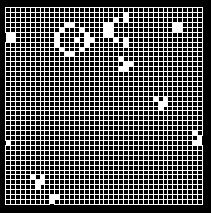
Laptop - Intel Core i7 9750H @ 2.6 GHz (4.5 GHz Turbo) – 6 core CPU with 12 threads. (As this was a laptop CPU, all battery settings were set to max and a charger cable was plugged in during testing).

Desktop – AMD Ryzen 7 2700X @ 3.7GHz (4.3 GHz Boost) – 8 core CPU with 16 threads.

Desktop – AMD FX-8350 Black @ 4GHz (4.2GHz Boost) – 8 core CPU with 8 threads.

Each of these machines test the exact same set of parallel and sequential design patterns. This is then used to compare how each of these versions of CPU core/thread design affect performance.

Scaling of Array Sizes



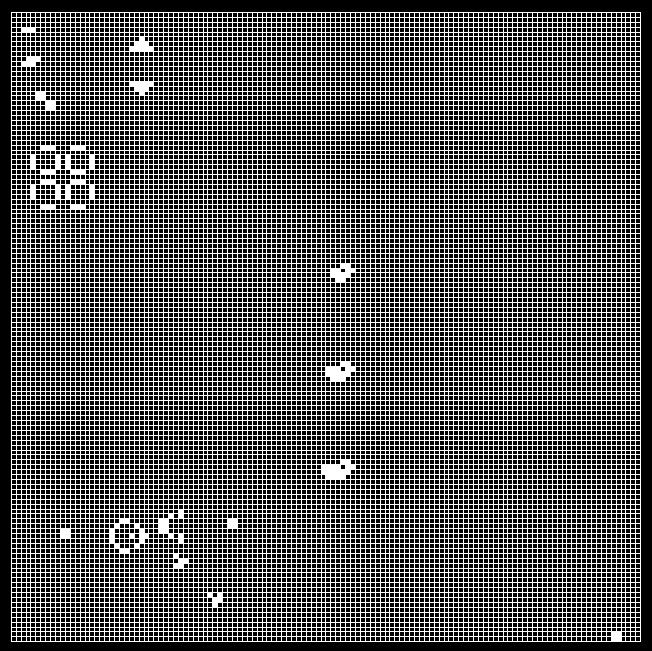
Examples of Testing Area Sizes (40, 160, 320)

This experiment will test how a nested and parallel\_for loop handles different sizes of 2D arrays. A single shape often used in the GoL, a Gosper Glider Gun, is shown in this experiment. This is a basic example of an indefinite design pattern that can continue to iterate across larger arrays. The purpose of this test is to see what size arrays benefit our sequential implementation and which benefit our parallel implementation.

The array sizes used for this are;

* 40 width x 40 height.
* 80 width x 80 height.
* 160 width x 160 height.
* 320 width x 320 height.
* An extra test of 240 x 240 was added after testing and will be discussed in the results section.

Oscillators + Spaceships



Various GoL Shapes over 128 Grid

This experiment will test various shapes common in the GoL, as well as numerous iterations of alive and dead cells spread across the whole of the 2d grid. Each shape has a required number of update loop steps it must go through before it completes and restarts its pattern. Therefore, it’s a suitable test to check for incompatibility between shapes and if each one completes its pattern properly whilst a number of other shapes iterate through their patterns.

First are Oscillators. These shapes are stationary and cycle through their patterns without much variation across the space of the grid.

The Oscillator shapes used are;

* A Blinker that completes its pattern every 2 iterations.
* A Toad that completes its pattern every 2 iterations.
* A Beacon that completes its pattern every 2 iterations.
* A Pulsar that completes its pattern every 3 iterations.
* A Penta-decathlon that completes its pattern every 15 iterations.

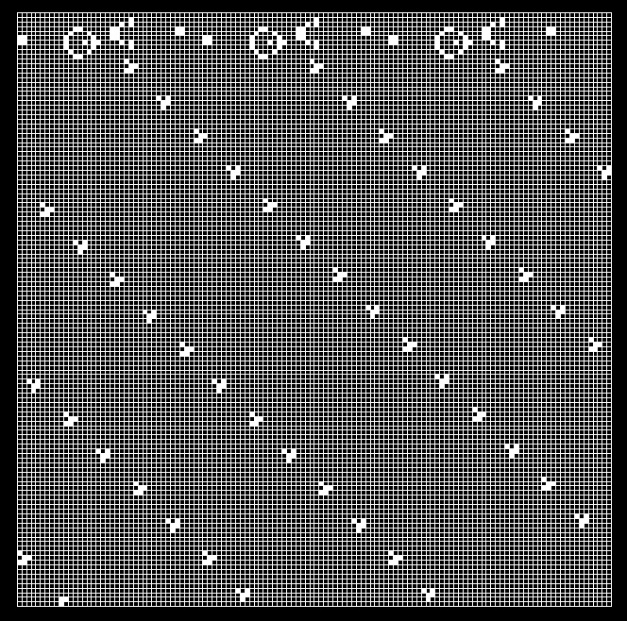
Next are Spaceships. These are defined by how they iterate across the grid as if they are moving. These are the most disruptive pattern due to them being able to crash into other shapes, resulting in unpredictable outcomes.

The Spaceship shapes used are;

* A Glider, moving southeast across our grid.
* A light-weight spaceship, moving east across our grid.
* A middle-weight spaceship, moving east across our grid.
* A heavy-weight spaceship, moving east across our grid.

These are tested on a 128 x 128 sized 2D array.

Indefinite Patterns



Three Glider Guns that continue to produce long strings of Gliders

As previous mentioned, a Gospel Glider Gun is one example of an indefinite design pattern, which grows indefinitely across the grid. Due to this, three guns are used in this test to simulate a stress test. These three shapes will continue to produce gliders over time, which should show some slowing of update iterations as time goes on.

These are also tested on a 128 x 128 sized 2D array.

A final note about these tests relate to boundary checking. In most iterations of the GoL, the nearby cell check will also account for the borders of our 2D array and won’t apply the ruleset if neighbour cells are off the grid. As all the tests include Gospel Glider Guns, these gliders in motion will iterate onto the next row or column and continue its trajectory. Because of the extra stress put on the CPU as these continue to iterate across the screen, these boundary checks have been left out across all the tests.

**Results & Reflection**

Scaling of Array Sizes

In the above graph, bars are used to represent the four CPUs and how quickly they completed their 600 update cycles. Across the first three sets of tests, the Intel 6-core 12 thread CPU clearly outperformed the rest. Then in our 240x240 array test, the Intel 4-core desktop CPU become closely comparative for parallel and won the test for sequential. However, this test was added due to the 4-core being unable to run at 320x320. On both tests, the program would crash. Another interesting result is how the AMD 8-core single threaded CPU manages to beat or keep up with the Intel 4-core in the early tests.

Overall, when comparing speeds, the Intel laptop CPU was the most reliable. This may be due to how our parallel threading library has been written and optimised for Intel CPUs, which could cause the AMD CPUs to struggle in speed. Also, having the extra 4 threads to spread over may reduce the load enough on the laptop CPU to be able to run the 320x320 array test over the 4-core. Both AMDs are very slow in the later tests, most surprisingly with the Ryzen. Comparing them, having the 8 extra threads over the single threaded CPU doesn’t seem to give it much of an edge at all in the tests.

One final note is that when it comes to our smallest data set, two of our CPUs run sequential faster than parallel. It is unclear why this is the case and could use testing on even smaller data sets. However, the test design pattern needed a minimum cell size of 35 to be drawn on our grid. This trend however quickly changes across the rest of the tests, showing the scalability of parallel as the data sets increase.

Oscillators + Spaceships

Oscillators and Spaceships Test

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Total time per 600 Update cycle (Nano-seconds)  Sequential Parallel | | Average time per Update cycle (Nano-seconds)  Sequential Parallel | |
| Intel 6 Core 12 Thread Laptop | 2042408600 nanoseconds (2.04 seconds) | 835692700 nanoseconds (0.84 seconds) | 3404014 nanoseconds | 1392821 nanoseconds |
| Intel 4 Core 8 Thread Desktop | 1691016147 nanoseconds (1.69 seconds) | 819651450 nanoseconds (0.82 seconds) | 2818360 nanoseconds | 1366085 nanoseconds |
| Ryzen 8 Core 16 Thread Desktop | 3809954900 nanoseconds (3.81 seconds) | 1310374400 nanoseconds (1.31 seconds) | 6349924 nanoseconds | 2183957 nanoseconds |
| AMD 8 Core 8 Thread Desktop | 4014415300 nanoseconds (4.01 seconds) | 1804184000 nanoseconds (1.80 seconds) | 6690692 nanoseconds | 3006973 nanoseconds |

For the next test, the time set for each update loop to complete has been plotted on a line graph to compare both average times and spikes in CPU load.

As this was a medium array-size test, predictably most of the parallel versions were faster. However, the intel 4-core was faster at running sequential than the AMD was at running parallel, albeit the number of high spikes in its pattern. This could highlight the difference in CPU architecture between AMD and Intel, as well as age. The AMD is around 10 years older than the Intel, so while their individual core GHz speed are comparable, age might be slowing the AMD down.

One last bit of data that stands out is how comparable both Intel parallel implementations are. While the previous array test showed the advantages of extra threads between these two CPUs, here we can see the difference between how differently a desktop and laptop CPU can be when it comes to power.

Indefinite Patterns

Three Glider Guns Test

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Total time per 600 Update cycle (Nano-seconds)  Sequential Parallel | | Average time per Update cycle (Nano-seconds)  Sequential Parallel | |
| Intel 6 Core 12 Thread Laptop | 2028570000 nanoseconds  (2.03 seconds) | 832944100 nanoseconds (0.83 seconds) | 3380950 nanoseconds | 1388240 nanoseconds |
| Intel 4 Core 8 Thread Desktop | 3576142660 nanoseconds  (3.58 seconds) | 1413938452 nanoseconds (1.41 seconds) | 5960237 nanoseconds | 2356564 nanoseconds |
| Ryzen 8 Core 16 Thread Desktop | 3843902100 nanoseconds  (3.84 seconds) | 1337413200 nanoseconds (1.34 seconds) | 6406503 nanoseconds | 2229022 nanoseconds |
| AMD 8 Core 8 Thread Desktop | 4006043900 nanoseconds  (4.01 seconds) | 1788868000 nanoseconds (1.79 seconds) | 6676739 nanoseconds | 2981446 nanoseconds |

This test also took the times for each CPU to complete each update cycle. This test had different levels of load over time, as more and more gliders were created by the glider guns.

Straight away, it’s very noticeable that the Intel 4-core on both tests never kept a steady load time, with some spikes on update loops being twice as slow as the next weakest CPU. This is surprising due to how it handled the previous shape test with a variety of design patterns. Again, the Intel 6-core was noticeably ahead compared to its rivals, with both tests having very smooth lines of distribution.

**Conclusion**

Overall, the breakdown of each CPU and how it performed on the test weren’t too surprising. Both AMD CPU’s fell behind in most tests, but it was more often than not the unpredictable performance of the 4-core Intel desktop CPU that skewed the data. As it is on a desktop and is one of the more recent generations of Intel CPU, I predicted it would be top on most of these tests. However, not being able to fully complete the array test as well as the scattered update times in the glider gun test meant that the AMD CPUs would be the choice for reliable scaling of parallel code on desktops.

It was also surprising how well the laptop performed even with its low base GHz speed. The tests were fairly quick, which didn’t give it much time to allocate extra power to the task. However, it was clearly the fastest even with its limitations, making this recent generation of hexacore CPU design the best choice for parallel code implementation when comparing all 4 options.

Finally, when it came to data sets that were anything bigger than 40 by 40 arrays, the parallel implementation used on each CPU was clearly better across every test. Each test showed the benefits scaling in parallel no matter the CPU and thread pool design. Therefore, when handling large data sets, implementing parallel code designs should make considerable time saves across the board.