The Game of Life

Concurrent Computing Coursework

Josh Felmeden, NK18044
Antoine Ritz, EV18263

December 4, 2019

Functionality and Design

Our solution was built up by initially creating a single threaded solution to the problem. This version iterates through the board bitwise, and for each bit gathers all the 'neighbours' for the cells (the 8 directly adjacent cells). From this, the logic is applied and the cell is updated if necessary. This is repeated for the desired number of turns.

From this, we created a multi-threaded solution. We split the board up into strips and passed each strip to a worker. However, each worker would also need information from the lines directly above and below its strip of cells (called *halo lines*). We decided to pass these halo lines wrapped around the strips, so that the workers are able to calculate each cell correctly. Once they have completed their strip, they return it to the `distributor` function. The function reconstructs the world and begins the process again for the desired number of turns.

One problem that we ran into was that we were passing the world by means of pointer. This led to problems due to premature changes being applied to the board. To solve this, we used two channels: `inputChans` and `outChans` (an array of channels with one channel for each worker) to pass the board to the workers.

The processing of the program is currently unable to be cancelled, and therefore we added the ability to quit, pause processing, and show the current state of the board with key presses. Alongside this, we also implemented an output of the number of alive cells every two seconds using a *ticker*.

Following this, we added the ability to allow the number of workers to support all multiples of two, rather than powers of two alone. Initially, this proved difficult, since at least one worker would receive a smaller strip than the others, and consequently meant that some workers would finish sooner than others. This resulted in a **deadlock**, because the world reconstructing function expected all strips to be of equal size, and therefore it attempts to read a strip of equal size for every worker. However, the final worker returns a smaller strip and so the `distributor` function is left waiting for more bits to arrive on the channel indefinitely. To resolve this, we made each worker apart from the final one work on the same number of lines, and the final worker simply had the remainder. Then, to reconstruct the world, we processed the final worker output separately from the rest.

Finally, passing the entire world between each turn is time consuming, because in reality, the only information that needs to be passed between the workers are the *halo lines*. Additionally, there is no reason to reconstruct the entire world each turn.

Implementing this proved to be problematic, as sometimes the workers would get out of sync with one another. To rectify this, we implemented a *master* thread, which served as a hard limit on the speed of the other threads to ensure that the threads would not exceed one another. We passed in a new structure called `flags` with a variable called `masterTurns` via pointer, which contains the current turn of the master thread.

We were unable to implement the user interaction via key press for this implementation

Tests, Experiments, and Critical Analysis

Stage 1a — Single Thread

Table 1: Benchmark comparison for Stage 1a

Benchmark	**Baseline result** (ns/100 turns)	**Our result**	**% Difference**
128x128x2-12	73689886	2021400726	36%
128x128x4-12	538915394	2010389496	26%
128x128x8-12	261671491	2009465802	17%

Table 2: CPU usage comparison for Stage 1a

Benchmark	**Baseline CPU usage**	**Our CPU usage**	**% Difference**
128x128x2-12	185%	100%	185%
128x128x4-12	298%	100%	298%
128x128x8-12	425%	100%	425%

This solution is the slowest because of the lack of multi-threading. The average bench is 135.048s.

Stage 1b — Divide and Conquer

Table 3: Benchmark comparison for Stage 1b

Benchmark	**Baseline result** (ns/100 turns)	**Our result**	**% Difference**
128x128x2-12	736946012	1704893591	43%
128x128x4-12	537751669	1273465510	42%
128x128x8-12	362735057	1004595662	36%

Table 4: CPU usage comparison for Stage 1b

Benchmark	**Baseline CPU usage**	**Our CPU usage**	**% Difference**
128x128x2-12	185%	188%	98%
128x128x4-12	298%	271%	109%
128x128x8-12	425%	348%	122%

By dividing the work up between the workers, this solution improves greatly on the single threaded solution, with an average benchmark of 90.460s.

Stage 2a — User Interaction

The benchmark results of stages 2a, 2b, and 3 are almost identical to that of stage 1b due to the lack of changes to the processing of the actual Game of Life logic. There are minor time increases for each stage due to additional constant processing. In this solution, there is a go routine that checks for a key input alongside the computation of the Game of Life logic which impacts the benchmark slightly. The average benchmark for this solution is 88.539s.

Stage 2b — Periodic Events

For this solution, there is a `ticker` that outputs the number of alive cells every 2 seconds. The impact on the overall runtime of the program is minimal, with the average benchmark being 94.175s.

Stage 3 — Division of Work

This implementation is the same as stage 2b in the case of the benchmarks, since the benchmark does not test for processes with threads that are not powers of two. The average benchmark for this solution is 90.832s; a minor improvement on stage 2b that is simply a result of the normal variation in program runtime.

Stage 4 — Cooperative Problem Solving

Table 5: Benchmark comparison for Stage 4

Benchmark	**Baseline result** (ns/100 turns)	**Our result**	**% Difference**
128x128x2-12	735620822	716510865	102%
128x128x4-12	530949831	483555762	109%
128x128x8-12	361530673	333119799	108%

Table 6: CPU usage for Stage 4

Benchmark	**Baseline CPU usage**	**Our CPU usage**	**% Difference**
128x128x2-12	185%	289%	64%
128x128x4-12	299%	425%	70%
128x128x8-12	425%	667%	63%

The final solution is significantly faster than all previous implementations by drastically reducing the necessary computations. The average benchmark is reduced by over 50% to only 41.679s.

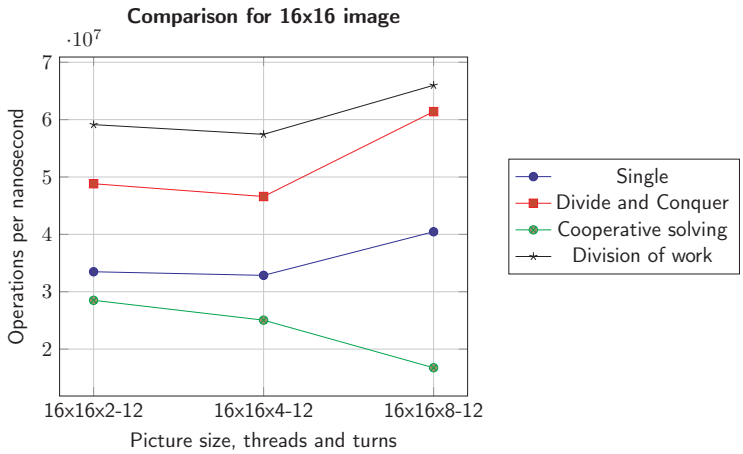
Conclusions

For smaller image sizes (Figure 1a), the single thread solution outperforms the initial divide and conquer method. This is because of the large overhead cost of splitting the image up and reconstructing the world after each turn. However, for larger images (Figures 1b and 1c), the divide and conquer algorithm does improve on the times set by the single thread, because the cost of reconstructing the world and splitting the threads is constant and does not rely on the size of the image.

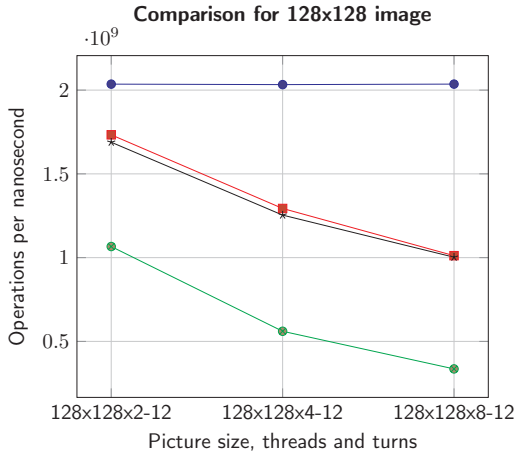
In general, the performance of the divide and conquer algorithm and the division of work algorithm is largely the same, especially for larger images. This is due to the fact that the only major difference between the two algorithms is that division of work algorithm allows the splitting of threads by multiples of 2, unlike the divide and conquer algorithm which splits the threads by powers of 2. This disparity between the two algorithms does not affect the performance of the algorithm and explains the minimal difference in the performance of the two solutions.

The cooperative solving solution is consistently faster than any other algorithm. The cooperative solving algorithm is always faster than both stage 3 solution because the cooperative solving algorithm does not reconstruct the entire world after each turn, and instead passes the halo lines to the worker above or below it. Another point where the cooperative solution saves time against the stage 3 implementation is the fact that there is no need to calculate the splits on every turn, since each worker retains the information of its split after every turn. Additionally, the workload of each worker remains almost exactly the same as the workers in the stage 3 implementation.

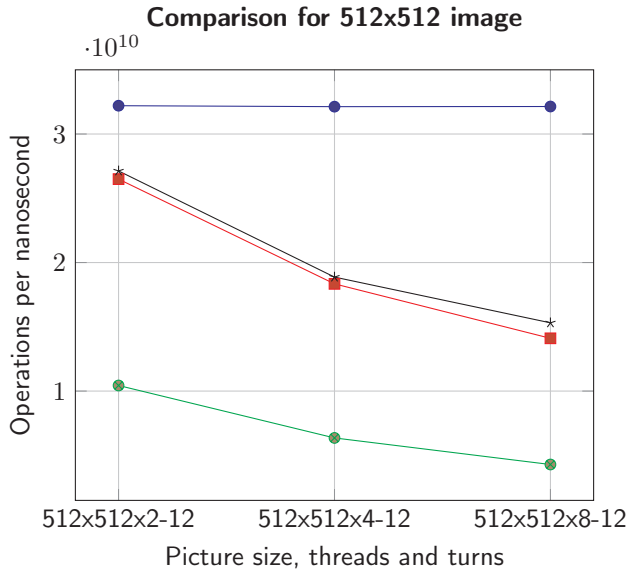
The cooperative solution vastly outperforms the single thread solution due to the ability to divide the work among the separate threads. The single thread must procedurally compute each bit of the world, while the cooperative solution is able to have each thread computing a bit at the same time.



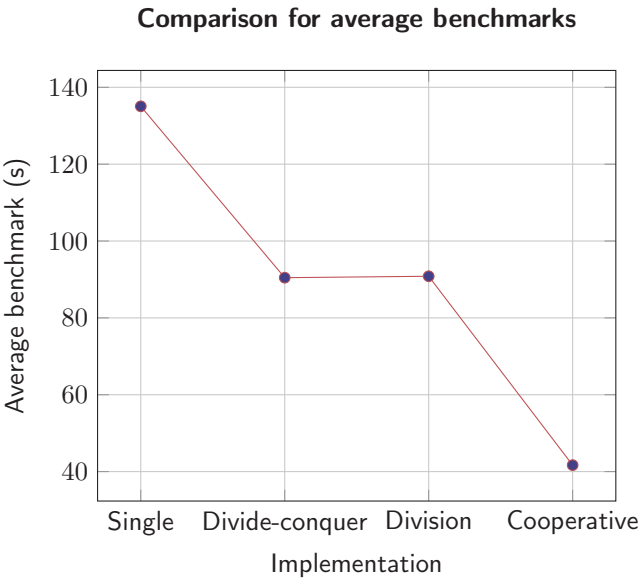
(a) 16x16 operations per nanosecond comparison



(b) 128x128 operations per nanosecond comparison



(c) 512x512 operations per nanosecond comparison



(d) Average benchmarks for each implementation

Figure 1: Comparison graphs