Parallel Conway's Game of Life

Stefanie Kozera
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
e-mails: skozera@gmail.com

***Abstract*—Conway's Game of Life is a cellular automaton algorithm - a set of rules run over a grid - which makes it a prime candidate for optimization using parallel computing [1]. This implementation of Conway's 'Life' utilizes Intel's Threading Building Blocks (TBB) library to parallelize the Life algorithm, specifically, parallel_for to optimize the execution of the main algorithm, parallel_reduce to count the number of alive cells in a given Game of Life generation, and parallel_invoke to run the prior two algorithms at the same time.**

This implementation features an option menu that allows the user to modify many parameters, such as game board size and generation count. By changing the parameters of the game it was found that with sufficient game board size there are indeed very significant gains to be had when parallelizing this algorithm. The biggest optimizations came from the parallelization of the main algorithm utilizing parallel_for, but the most efficient configuration utilized all three parallelization algorithms working together. Conway's Life is a strong contender for optimization by parallel computing, provided the speed increases are wanted and achievable for its given implementation.

I. Introduction

In 1970 a Cambridge mathematician named John Horton Conway released his carefully tuned cellular automaton algorithm - the Game of Life. Cellular automaton refers to systems where rules are applied to cells within a regular grid, and Conway's specific algorithm was not very complicated, only a few rules explaining which of two states the cells fall into, alive or dead [1]. None-the-less, these simple rules result in myriad patterns, and mathematicians and other enthusiasts are still finding new patterns and start-states to this day. Since 1970, the Game of Life has developed something of a cult following, and it is very likely one of the most programmed computer games in existence, due to its interesting results and relatively simple ruleset [1].

An algorithm applied over a 2D grid is the sort of problem that thrives within parallel computing. This project is a proof of concept. If one was to run Life as fast as they possibly could, would there be significant gains utilizing parallel computing with a library such as Intel's TBB? In addition, as a proof of concept, it was important that the implementation be easily displayed to interested parties. This implementation of Life features a user interface designed with the ncurses library, a handy library for interacting with the terminal [2]. There is an options menu, which allows the user to modify many different parameters for the game, and a demonstration mode, so the user can see the Game of Life running at a speed the human eye can easily parse, before utilizing the parallelization strategy to push the algorithm to the limit.

While Conway's Game of Life is very interesting, and still incredibly popular many years after its inception, it is not the most useful of algorithms. However, cellular automata do have real world applications. There are models used in the fields of physics for gas and fluid dynamics, cryptography applications, and models used to study epidemiology, anthropology, and biology [3]. A parallel implementation of the Game of Life proves that these other models might find performance gains in their implementations if they choose to go with a parallelized approach, which could save significant amounts of time when used for highly complicated algorithms.

II. Methodology

A. The Game of Life

The Game of Life has the following rules [4]:

1. A live cell with fewer than two neighbors dies
2. A live cell with two or three neighbors lives
3. A live cell with more than three neighbors dies
4. A dead cell with exactly three neighbors lives

Consider the following example in figure 1, where '0' indicates a dead cell and '1' indicates an alive cell. We will consider this current state generation 'i' or gen[i]. The middle square in a 3 by 3 grid will be considered element 'y,x' of a 2D array, or element[y][x].

Fig 1: gen[i]

1	1	0
1	0	0
0	0	0

Considering only the middle cell, element[y][x], when we apply the Life algorithm, all eight surrounding cells are taken into account. The current element[y][x] is dead, but a

dead cell can become an alive cell if exactly three of its neighbors are alive, which is the case in figure 1. We will apply the following logic:

```
neighborCnt = element[y-1][x-1] + element[y-1][x] + element[y-1][x+1] + element[y][x-1] + element[y][x+1] + element[y+1][x-1] + element[y+1][x] + element[y+1][x+1]

if( gen[i].element[y][x] == 0 ) {
    if ( neighborCnt == 3 ){
        gen[i+1].element[y][x] = 1
    }
}
```

Specifically, we add up all 8 surrounding cells, then determine if our current middle cell, element[y][x], is alive or dead. If it is dead, we then see if there are 3 living neighbors and if so this dead element[y][x] becomes alive. Figure 2 shows the results in gen[i+1].

Fig2: gen[i+1]

x	x	x
x	1	x
x	x	x

Note, only the middle cell's state was determined for gen[i+1]. All surrounding cells must undergo this algorithm on their own to discover their own subsequent state.

Let's take another example, where element[y][x] is alive and has many alive surrounding neighbors, in figure 3.

Fig 3: gen[j]

1	1	0
1	1	0
0	0	1

An alive cell is guaranteed death if it has more than three living neighbors, which is the situation element[y][x] finds itself in. In this case the following algorithm would be appropriate.

```
else if( gen[j].element[y][x] == 1 ) {
    if ( neighborCnt > 3 || neighborCnt < 2) {
        gen[j+1].element[y][x] = 0
    }
}
```

This time, we check to see if element[y][x] is alive, and if it has the appropriate number of living neighbors. As it does not, during the next generation, gen[j+1], this cell will be dead and set to 0. See figure 4.

Fig 4: gen[j+1]

x	x	x
x	0	x
x	x	x

This logic forms the basis of all Conway's Game of Life implementations. One problem to consider is the situation of edge cases. In the algorithm as originally imagined by Conway, there are no formal edges. The surrounding grid of dead cells reaches out infinitely in all 2 dimensional directions, with the only consideration being any live cells. This was not appropriate for this configuration, as one of the adjustable parameters is the board size, and having an infinite board will not provide as clean results when determining the usefulness of the parallelization strategies.

Instead, the board was confined to a 2D array of a particular size, and on an edge the same 4 rules are applied. An edge case will simply have less potential neighbors to count. For example, a normal cell has 8 neighbors to check, but a case on the corner will only have 3 neighbors to check, and a case along one of the edges will only check 5 neighbors.

B. 'This' Game of Life

The ncurses library was used both to create an options menu, and to facilitate a demonstration mode to display Life in a visual way. The menu options are as follows:

1. Mode: Demo/Calc
2. Board Size X
3. Board Size Y
4. Starting State
5. Generations
6. Parallel For On/Off
7. Parallel Reduce On/Off
8. Parallel Invoke On/Off

This report serves as documentation for this Life program. Functionality of the menu options is as follows.

Mode: Demonstration/Calculation. Demo mode will run an instance of Life within the terminal, with each generation showing for 500ms. The selected board size, generations, and parallelization strategies are not utilized in demo mode, and the program will run until the user exits. Calc mode will use all selected parameters to run an instance of Life until completion, then will provide information to the user within a menu. The information will consist of the board size, generations, the final alive count, which parallelization options were enabled, and the time to complete the Game of Life.

Board Size X: The size of the board along the X-axis. Type in the requested value and press enter, or backspace to leave.

Board Size Y: The size of the board along the Y-axis.

Starting State: Determines the starting state of the board. Percentage based will fill the board at a 25% density across the entire board size. Preconfig1 is a Gosper Glider Gun starting state which will infinitely fire Glider Guns into the bottom right corner of the board [5]. Preconfig2 and Preconfig3 are not used.

Generations: The number of generations to run in calc mode. Type in the requested value and press enter, or backspace to leave.

Parallel For: Enable or disable the parallel_for algorithm. The option with the asterisk is the option that is used.

Parallel Reduce: Enable or disable the parallel_reduce algorithm.

Parallel Invoke: Enable or disable the parallel_invoke algorithm.

Within the main menu, selecting start will begin the program, and pressing exit will exit the program without running the Game of Life.

In addition, when running calculation mode, the count of alive cells during all generations will be recorded in an array, and then output to a .txt file named AliveCnt.txt. This file can be used to verify functionality of the program, and is interesting information about the Game of Life throughout the generations it has run.

C. *Parallel Game of Life*

This implementation utilizes three parallelization strategies to test the optimization gains during Life. They are as follows:

1. Parallel For: Used when applying the Game of Life algorithm across the entire board. Parallel_for is also used to save the 'Next Generation' variable to the 'Current Generation' variable, to prepare the variables for the next generation calculation. When enabled or disabled, both uses of parallel_for are enabled or disabled. It is not possible to only enable one or the other.

2. Parallel Reduce: Used to count the number of alive cells on the game board for each generation. These results are output to AliveCnt.txt, and the final count is displayed in the terminal.

3. Parallel Invoke: Used to run the Game of Life algorithm and the alive count at the same time in parallel.

All parallelization options are individually toggleable. For example, it's possible to only utilize parallel_invoke, in which case the Game of Life algorithm and the alive count would both run in parallel, but neither would utilize parallel_for or parallel_reduce, and instead would be handled using sequential logic. Likewise, it's possible to enable parallel_for and parallel_reduce, but not parallel_invoke, in which case the alive count and the Life algorithm would both be run in parallel, but would run one after another, not at the same time. A standard sequential operation can be utilized by disabling all parallel strategies.

III. Experimental Setup

This program was run on Ubuntu 24.04.1 LTS, on a desktop computer with an AMD Ryzen 9 5900X, with 12 cores, 24 threads, and a base clock speed of 3.7 GHz. It was compiled and run in GNOME Terminal, version 3.52.0, which is the terminal included in the Ubuntu version used. When taking time measurements, no other programs were running. The TBB version used was 2021.11.0-2ubuntu2. The ncurses version used was 6.4+20240113-1ubuntu2.

It was expected that there be some performance gains when utilizing parallelization. The question was how large would these performance gains be, and would changing any of the various other factors affect these results. All timing results are averages of 10 attempts.

IV. Results

For a baseline understanding of the optimization results, a board size of 500x500 was utilized, over 50 generations. So results were consistent, preconfig1, the glider gun, was used. Then, all combinations of parallelization options were tried and recorded. See Table 1. X indicates the algorithm was activated, where a blank square indicates the algorithm was not activated.

Table 1:

For	Reduce	Invoke	us
			112036
X			18305
	X		121199
		X	224865
	X	X	218305
X		X	15731
X	X		16563
X	X	X	15658

Using this data, we can compare all three parallelization strategies, both in isolation, and when used in conjunction with other parallelization strategies.

Parallel_for is clearly the most useful strategy employed. All results that utilize parallel_for are significantly quicker than use without, sometimes by a factor of 10. Our baseline sequential results is 112,036 us. Use of parallel_for reduces this to 18,305 us. It is clear that the main algorithm is a significant bottleneck. This makes sense, as parallel_for is used twice in the program, and is 'doing the most' in terms of calculation. There are many comparisons it has to process to determine the next generation, and then it has to be used again to set the variables so it can repeat the process

Parallel_reduce does not result in performance gains when used on its own, but interestingly, it does result in performance gains when used with parallel_for. This result seems a little odd, and is perhaps due to the differences in implementation when using parallel_for versus not. It was not possible to completely isolate the parallelization strategies, as setup differs when using them compared to the standard implementation.

Parallel_invoke goes the same way as parallel_reduce. When using parallel_invoke on its own, the results are very unfortunate, taking almost double the time of the sequential implementation. Again though, if parallel_invoke is used with parallel_for, there is a decent gain of around 300 ms. What is interesting is the results of all three algorithms compared to the results while utilizing just parallel_for and parallel_reduce, with a time of 15,658 ms vs. 15,731 ms. Is there any use in running parallel reduce when utilizing all three algorithms? This perhaps points to the parallel_for bottleneck again. If the main Life algorithm is taking longer than the alive count algorithm, even without parallel_reduce, we would not see any performance gains, as invoke would not be able to speed up faster than the main Life algorithm.

Whether parallel reduce is actually useful with parallel_for and parallel_invoke, or not, using all three algorithms did result in the quickest time by a little bit, so moving forward we will compare all three algorithms on, and all three algorithms off, as we adjust the other parameters available in this Life implementation. Next, we will try a board size of 2000x2000, to see if a larger board results in more parallelization gains. The results are located in Table 2.

Table 2:

for/reduce/invoke	Board Size	us
	500x500	112036
X	500x500	15658
	2000x2000	1820136
X	2000x2000	207229

It is clear even with a board size of 2000x2000, there are still performance gains when using parallelization, but how much better is the performance in comparison with a sequential implementation? To analyze this, we will take the parallel time as a percentage of the sequential time using the equation (para_on)/(para_off) * 100%. Results are in Table 3.

Table 3:

Board Size	Time % of Seq
500x500	13.97
2000x2000	11.39

Looking at these results, at 500x500, the parallel operation takes ~14% of the time of the sequential operation. At 2000x2000, the parallel operation takes ~11.4% of the time of the sequential operation. This is indeed an improvement, and shows that as the board gets significantly larger there are more optimization improvements to be had.

What if we did the same, but with generations? See Table 4.

Table 4:

for/reduce/invoke	Generations	us
	10	22007
X	10	4157
	25	54341
X	25	8469
	50	112036
X	50	15658

And as percentages, in table 5.

Table 5:

Generations	Time % of Seq
10	18.89
25	15.58
50	13.97

We have similar results here. As we increase generation count, the percentage of time taken decreases. This is likely due to a certain amount of overhead in starting and ending the program. With a larger board, or more generations, there is more opportunity for the parallelization algorithms to impart improvement.

One last lever to pull: the starting state. This implementation contains two starting state options, a glider gun configuration that is rather small in comparison to the rest of the board, a 36 by 9 2D array, and a random percentage configuration that covers the entire starting board in alive cells at a random density of 25%. Would the two different starting configurations result in different timings, keeping board size equal at 500x500, and generations equal at 50? See table 6:

Table 6:

for/reduce/invoke	Configuration	us
	Glider Gun	112036
X	Glider Gun	15658
	25% coverage	167445
X	25% coverage	19987

At a percentage based coverage, it does indeed take longer to execute the Life program, in both a sequential and a parallel approach. If we run our percentage equation again we see the following results in Table 7:

Table 7:

Starting State	Time % of Seq
Glider Gun	13.97
25% coverage	11.94

We obtain better time percentages during the percent coverage scenario, but note that the 25% percent coverage scenario takes longer in both a sequential and parallel approach. It might be that with such a high coverage density, with many cells changing states across the whole board, there is a bigger slowdown in the sequential implementation that is partially mitigated through the parallel implementation. Another reason the parallel implementation works very well for the Life algorithm.

These results overall seem largely expected. The Game of Life is a very good candidate for parallelization due to the fact that it is an algorithm run over a very large set of variables, a common example for use with parallel_for. It makes sense then, that parallel_for netted the most time improvements in isolation, although it is interesting that all three options enabled at once provide the most improvements overall. Once the main Life algorithm is handled, it seems there is plenty of opportunity for smaller incremental improvements that can be made with creative application of other parallelization techniques.

Conclusions

Conway's Game of Life, and other cellular automata, are excellent candidates for parallel computing. This implementation of Life proved that parallelization can create incredible time saving benefits, by factors of 10 or beyond, depending on the scope of data being manipulated by the cellular automaton algorithm. This, of course, depends on the purpose of the particular cellular automaton. Conway's Game of Life is often experienced visually, which was why

this implementation featured a specific mode where the user can watch the algorithm with timing easy to parse via human eye, but other cellular automata serve different purposes such as cryptography and biology [3]. In these cases, optimizations from parallel computing could save huge amounts of time.

This implementation still has space for improvement. The menus are not very robust, and could use some cleaning up. In addition it would be good to add more pre-configurations, or the ability to load in any preconfig from a defined file structure. There are a few popular ones available for the Game of Life already. In addition, it would be interesting to see if the performance gains are similar when using a different parallel computing library, such as Pthreads. The hope would be that this implementation could serve as a basis for other cellular automata algorithms; ones with different uses, both algorithms that are amusing, and algorithms that serve important, real world applications.

References

[1] P. Callahan, "What is the Game of Life?," Math. http://www.math.com/students/wonders/life/life.html (accessed Dec. 14, 2024).

[2] P. Padala, "NCURSES Programming HOWTO," *Tldp*, Jun. 20, 2005. https://tldp.org/HOWTO/NCURSES-Programming-HOWTO/intro.html#WHATIS (accessed Dec. 14, 2024).

[3] R. Awati, "Cellular Automaton (CA)," *TechTarget*, Dec. 2021. https://www.techtarget.com/searchenterprisedesktop/definition/cellular-automaton

[4] "Conway's Game of Life," Conwaylife, Dec. 02, 2024. https://conwaylife.com/wiki/Conway%27s_Game_of_Life

[5] "Gosper Glider Gun," Conwaylife, Nov. 21, 2023. https://conwaylife.com/wiki/Gosper_glider_gun