# Conway’s Game of Life in Parallel - FreeST

Diogo Soares and Marta Correia

Departamento de Informática da Faculdade de Ciências da Universidade de Lisboa
{fc44935, fc51022}@alunos.fc.ul.pt

**Abstract.** The main goal of this project is to present a parallel way of implementing the Game of Life Algorithm in the functional programming language FreeST. This paper is divided into three main parts. We first explain the Game of Life algorithm. Secondly, we present the way this algorithm was implemented in two parallel and one sequential version. Lastly, we compare the versions, concluding that, in this language, the sequential version is faster than the parallel one, and draw further conclusions as to why.

**Keywords:** FreeST, Parallel Programming, Game of Life, Cell Automation

## 1 Introduction

When dealing with a program that requires a very large number of computations, that is, that are computationally intensive, its necessary to find a way to get the results in a faster, more efficient way. That is why, the studying and development of methods to parallelize programs, in a way that the tasks that can be done at the same time, are executed in parallel, is a very important topic to achieve optimization and scalability.

Considering this, our goal with this project was to: create a parallel method to implement Conway’s Game of Life algorithm; provide an implementation for such method, using a recently developed programming language, FreeST; and make a comparison between the parallel and sequential version, also developed in FreeST.

## 2 Background

The Game of Life algorithm is a zero-player game designed by John Horton Conway in 1970. The purpose of the game is, by giving an input, to create an interesting and unpredictable cell automation that evolves independently through a set of defined rules.

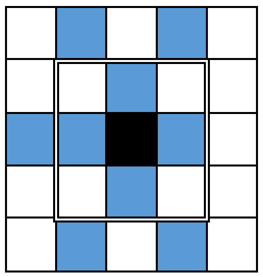


Figure 1 - Neighbors of the cell in black are the 8 cells around it

The universe of Game of Life is a two-dimensional orthogonal grid of square cells. These cells are in one of two states: live or dead. Each cell interacts with eight neighbours that correspond to the cells immediately above, below, to the right, to the left and each diagonal of the cell (Figure 1). At each new generation, to determine if the cell is alive or dead, the following rules are applied:

- If an alive cell has less than two neighbours alive, it dies of underpopulation.
- If an alive cell with two or three neighbours alive, it lives.
- If an alive cell has more than three neighbours alive, it dies of overpopulation.
- If a dead cell has exactly three neighbours alive, it becomes alive by reproduction.

## 3 Approach

We have developed three programs for the implementation of this algorithm: a parallel program, that for each row in the matrix, creates a new thread that deals with that row; a parallel program that takes more into account granularity, and that creates only *n* threads, being *n* the number of cores of the machine the program runs on; and a sequential version to be tested against the parallel versions and that runs only in one thread.

The base implementation is the same for all three programs. The matrix is iterated row by row, and for each row, we calculate two lists that contain the number of neighbours, left and right, of each cell. One of the lists, however, also counts each cell as a neighbour of itself while the other does not, as explained in Figure 2.

The way the recursion is set up, since this is a functional language, is that each call sends the list counting with itself as a parameter to the next recursion and receives from that call also a list but from the row below. Thus, each iteration will have the neighbours from the row above as a parameter and the neighbours from the row below as the return from doing recursion.

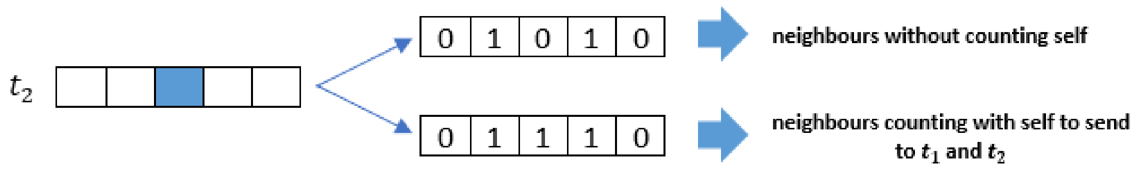

$t_2$
0 1 0 1 0
neighbours without counting self
0 1 1 1 0
neighbours counting with self to send to $t_1$ and $t_2$

Figure 2 – The two lists of neighbors, left and right, in which one counts the cell itself as well

With both lists from the rows above and below and with the second list of neighbours, without counting itself, the current iteration can get a list of the eight neighbours around each cell by summing the three lists together. After the sum, we can apply the game of life rules, as explained in Figure 3.

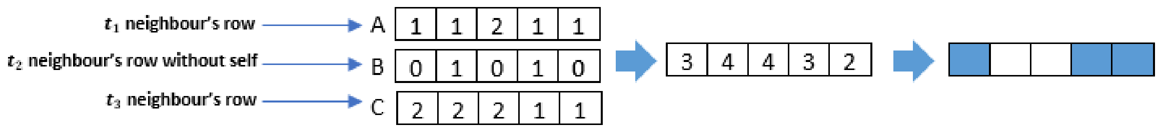


Figure 3 – The sum of the three lists of neighbors and the application of the game of life's rules, being the blue squares, alive cells, and the white squares, dead cells

From the recursion, we also received the bottom world where the rules were already applied so that we can concatenate the current row, after the game of life rules are applied, and the ones below. At the end, we'll have the various rows concatenated and a full new generation created.

## 4 Implementation Details

As explained in the last section, we implemented three versions for this algorithm, two of which are parallel and the third, a sequential version. We'll start by explaining the common structures the three of them use and then do a walkthrough through for each of them.

To test different worlds and number of iterations, the variables in any of the versions' *main* method can be altered. As default, we set up 10 iterations, in a world of 100 elements, where each row contains 10 elements.

To represent the world, instead of a 2D matrix, we had to create a linked list **World** that is either a **Nil**, if at the end of the world, or a **Tile**. Each **Tile** represents a cell in the matrix and consists of the index of the cell, its state (true if alive, false if dead) and the next **Tile** in the list.

We have also created a linked list **IntList** that represents the list of neighbours of each row. The **IntList** is either a **Nul**, if at the end of the list, or a **Number**, which contains the number of neighbours of that cell and the next **Number** in the list.

In FreeST, to make threads communicate, it's necessary to use channels, where each thread will have either a write or a read end to that channel. To make use of the channels in this language, we must break the objects, that we pass through them, into primitive types. With that in mind, we defined two types of channel: one that represents a collection, in which the **WorldChannel** and the **IntListChannel** fall into, representing respectively a world and a list of number of neighbours; and one that represents each object of the collection, which would be the **TileChannel** and the **NumberChannel**, that represent a cell and a number of neighbours, for that cell, respectively.

To initialize the world, the function *initWorld* was implemented. This function receives as a parameter the number of elements to create for the matrix. For each element, it creates a **Tile** with the current index, a random state given by function *multiplesThree* (that returns true if the index is a multiple of three) and the next **Tile** that comes from the recursion of the function. Another function could be given to randomize the state of the initial cells if provide in this method, instead of *multiplesThree.*

All three algorithms start by creating a new world and printing it. They then proceed to call the function *generations*, that is responsible for, given the number of iterations, calling the function *splitWork* to generate the next state of the world and printing it in the console.

### 4.1 Parallel Implementation

As also explained prior, we implemented two versions for the parallel implementation, in which one creates a new thread for each row, and the other has more into account granularity, dividing the matrix into *n* threads, being *n* the number of cores of the machine where the algorithm is being executed. The number of threads needs to be defined manually in the variable *cores* in the *main* function. We'll start by explaining the former.

### 4.1.1 Threads for each row

The function *splitWork,* in this version, will iterate through the world, row by row, using the function *splitRow* to split each row.

In each iteration, the algorithm creates two channels, **IntListChannel,** to pass information between threads (the thread for the row above and the thread for the row below), and a third channel, **WorldChannel,** to communicate again with the thread of the row above.

Each thread will then need five ends of channels: two to read the neighbour's list from the rows above and below; two to write its own list of neighbours to those threads; and one to write the world it created to the thread above. To make this work, we made it so that, in each recursive call, we send to the next iteration a channel end, **read2**, so the thread of the row below can read the list of neighbours this row is going to send. From the recursion, the current row receives two end channels, **bottomRead** and **bottomRead2**, one to read the list of neighbours of the row below, and the other to read the already calculated remaining of the world. As a result of recursion, this current row also receives, as a parameter, the end channel to read the list of neighbours of the row above, **topRead**. This way, this execution works as explained in the Approach section: each thread will receive the **IntList**s from the row's above and below, send its list to the both of them, and then return, to the top row, the concatenated world.

Following the recursion call, the program finally forks the process and sends the row to the *subserver* function, which will deal with the communication between the threads and the application of the rules as explain in the Approach section. At the end, it returns two channel ends, **read** and **read3**, so the row above can get this row's list of neighbours and the world, after the game of life rules are applied.

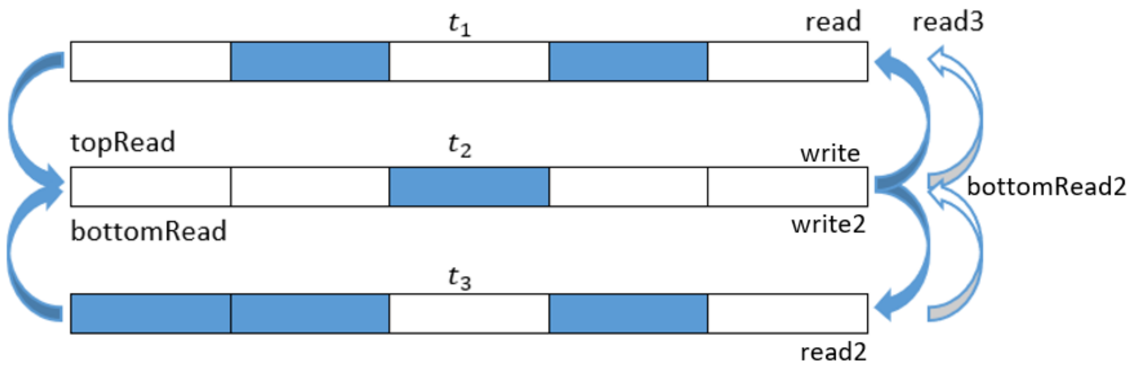


Figure 4 – Representation of the communication between the current thread and the threads that run the row below and the row above

### 4.1.2 N- Threads

For the version that considers granularity, the algorithm does basically the same process as the former, but instead of iterating by row, it receives a variable *cores,* that represents, the number of cores the program will run on. It then partitions the matrix in *n* parts, so that each process can run one of the partitions. Inside the partition, the algorithm runs as the sequential version.

In terms of communication, it uses the same channels as the previous version, but these communications are only done between the top and bottom row of each partition and the partition above and below it.

## 4.2 Sequential Implementation

The sequential implementation is simply an implementation of the algorithm explain the Approach Section. It uses only recursion, running in one thread and not needing any communication or use of channels to operate.

## 5 Evaluation

### 5.1 Experimental Setup

The machine used to test the three algorithms it's a computer with 16GB of RAM, an Intel®Core™ i7-6500U CPU @ 2.50 GHz and 4 logical processors.

### 5.2 Results

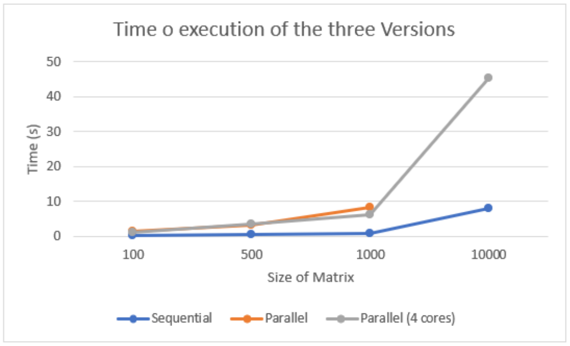
To compare the run times of each version, we tested various sizes and row size for the matrix, for some of which, we were unable to gets results because of the amount of time it took to run. For the parallel version with granularity, we have also tested changing the number of *cores* used, while still running on the same machine. The results are the following:

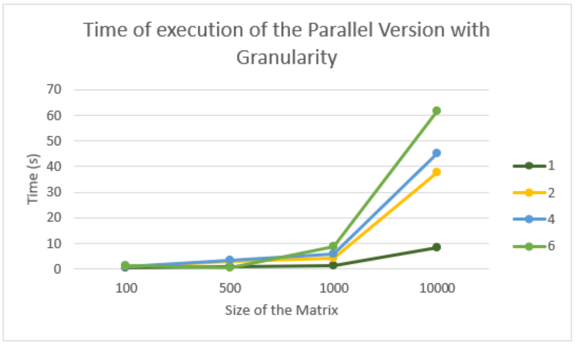
| | | | Time (s) | | |
|---|---|---|---|---|---|
| Iterations | Size | Row Size | Sequential | Parallel | Parallel (4 cores) |
| 10 | 100 | 10 | 0,127 | 1,302 | 1,119 |
| 10 | 500 | 100 | 0,426 | 3,221 | 3,549 |
| 10 | 1000 | 100 | 0,772 | 8,378 | 6,052 |
| 10 | 10000 | 100 | 7,938 | - | 45,346 |

Table 1 – Results of testing different sizes and row sizes in the sequential and the parallel version without granularity

| | | | Time (s) | | | |
|---|---|---|---|---|---|---|
| Iterations | Size | Row Size | 1 Core | 2 Cores | 4 Cores | 6 Cores |
| 10 | 100 | 10 | 0,562 | 0,956 | 1,119 | 1,285 |
| 10 | 500 | 100 | 0,839 | 2,948 | 3,549 | 0,794 (5) |
| 10 | 1000 | 100 | 1,259 | 4,363 | 6,052 | 8,963 |
| 10 | 10000 | 100 | 8,304 | 37,731 | 45,346 | 61,464 |

Table 2 – Results of testing different sizes and row sizes using the parallel version with granularity and a different number of cores





Graphic 1 – Comparison of time of execution of the sequential, parallel, and parallel with granularity (4 cores) versions

Graphic 2 – Comparison of time of execution by using different amount of partitions in the parallel with granularity version

## 5.3 Discussion

As we can see from the presented results, the sequential version was shown to be a lot faster that both the parallel versions. We can also conclude that, the more *cores* we use in the parallel version with granularity, the more the time it takes to execute, on the same machine.

Given that both the parallels and the sequential version diverge from the same base algorithm, as explain in the Approach section, the reason why we believe the sequential version ended up being much faster than both the parallel ones is that the time it takes for threads to communicate and transfer data between them, ends up consuming more time than just normally iterating through the world sequentially.

As explained previously, to use the channels to communicate between threads, in this language, is necessary to break each object into primitive types. By doing so, it forces the algorithm to recursively iterate through the object each time it needs to provide it to another thread. The same happens at the other end of the channel when the object needs to be reconstructed. Another factor that could contribute to this delay is the fact that the server, even if it runs a specific iteration faster then the client, it will be block in the next iteration while waiting for the client to send data again. These results are also visible when the complexity of the algorithms was calculated, observable in Table 3

| Algorithm | Complexity |
|---|---|
| Sequential | $O(i * n * (6r - 1))$ |
| Parallel | $O(i * s * (11r - 1))$ |
| Parallel with Granularity | $O(i * (s + 12r - 1))$ |

i – number of iterations
n – number of rows
r – size of rows
s – size of the matrix

Table 3 – Complexities of the three versions of the algorithms

In terms of the variation of time when using different number of partitions, the justification is the same. Since we use the sequential algorithm inside each partition, the less partitions we have, the less communication between threads is needed, thus making the algorithm much faster.

## 7 Conclusions

As we concluded in the discussion section, it wasn't possible to, using this programming language and the method we developed, to originate a parallel version of this algorithm that is faster than the sequential version presented, with or without granularity.

A way to determine if the method we presented could be feasible would be to implement it in another programming language. This could be tried out on both a functional language, such as Haskell, in other to test if the issue was dealing with recursion, or a more object orientated language, such as Java, that allows shared memory and a more efficient way of dealing with multiple threads.

For the implementation in Haskell, it would be interesting to test if we had access to native arrays instead of having to recursively iterate through the world every time, if the performance would've been different. For the implementation in Java, this could provide an interesting topic of study by comparison of the use of Share Memory versus communication between individual threads.

## Acknowledgments

Diogo Soares started by implementing a sequential version (a very bad one), while Marta Correia started by understanding and creating the channels. After the initial individual exploration of the language, we both gathered to discuss and draft the first implemented of the parallel version of the algorithm, which was the on without granularity. We both partook in the discussion and came up the final algorithm, which suffer some trims along the line, and the development of the second parallel version, as well as the sequential version. We also both did the report and spent around 40 hours, each, on this project, where 36 of them were together.

## References

1. 7.3: The Game of Life - The Nature of Code. (2015, August 10). [Video]. YouTube. https://www.youtube.com/watch?v=tENSCEO-LEc

2. https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life

3. Almeida, B., Mordido, A. and Vasconcelos, V.T., 2019. FreeST: Context-free Session Types in a Functional Language, In PLACES 2019, EPTCS, vol 291, pp. 12-23.

4. FreeST: Context-free Session Types in a Concurrent Functional Language, talk at Programming Languages for Distributed Systems, Shonan, 2019.

5. http://rss.di.fc.ul.pt/tryit/FreeST

6. http://rss.di.fc.ul.pt/tryit/CFSTEquiv