Life

Mark Mazel (mm4764)

December 2023

1 Introduction

This report aims to describe a parallel functional implementation for Conway's Game of Life. Conway's Game of Life is a well known cellular automata. Conway's Game of Life is the name of the algorithm which describes the state transitions of a 2D (often toroidal) grid of bits which represent cells which are either Alive or Dead. By the rules of Conway's Game of Life, the next state of each cell on the grid can be derived as a function of the prior state of the cell in question as well as the prior state of every cell adjacent to it (eight total). From Wikipedia:

1. Any live cell with fewer than two live neighbours dies, as if by underpopulation.
2. Any live cell with two or three live neighbours lives on to the next generation.
3. Any live cell with more than three live neighbours dies, as if by overpopulation.
4. Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction

2 Rationale

As seen above, given a grid of cells, the next state of a cell can be found as a function of 9 datapoints, the prior state of the given cell, and the prior state of its 8 adjacent cells. Because of this, this problem is inherently quite parallel and predisposes itself to a parallel solution. The problem also gives us a couple input parameters to play with:

1. Starting State
    - Although many configurations have been studied and there are shapes which have been studied extensively and named in the Game of Life community, for this Haskell approach, the input does not actually

matter from a performance standpoint at a high level. Aside from any optimization or caching going on, calculating the N^{th} step of an $X \times Y$ grid will take the same amount of function calls regardless of the original state of that grid.

2. Grid Dimension
    - The size of the input grid can be selected such that the overall runtime of the program is fast enough as to be reproducible without a large wait but slow enough to allow any speedup during parallelization to be easily noticeable.
3. Steps
    - The amount of steps to simulate can be increased if needed to balance the relative time of reading the input and dumping the output with the time of "actual" computation.

3 Algorithm

3.1 Overview

There exists an algorithm proposed in the 1980's called Hashlife which utilizes a hash-based memoization approach to more quickly be able to compute the future state of the grid at the cost of larger memory usage. For this report, I will focus on the straight-forward brute-force approach which calculates the state of a cell at a given step as a recursive tree of calls to the states of that cell and its adjacent ones.

3.2 Haskell Approach

For the Haskell approach to this, I adapted the approach discussed in this blog. As an input, it takes in a file describing the initial grid state where each character represents a cell with an $\times$ being a live cell and a $\cdot$ being a dead cell.

The parameters are listed as toplevel constants within the Main.hs file:

```
wIDTH :: Integer
wIDTH = 1000

hEIGHT :: Integer
hEIGHT = 1000

sTEP :: Integer
sTEP = 1
```

The grid is represented with the following data structure:

```
-- | A point in the grid
type Point = (Integer, Integer)
-- | The status of a cell
data Cell = Alive | Dead deriving Eq
-- | Description of a grid
type Grid = Point -> Cell
```

The initial grid is parsed from an input file specified via the command line with the following code:

```
parseInitialGrid :: B.ByteString -> Grid
parseInitialGrid s = go
  where
    ls = B.split '\n' s
    maxY = fromInteger wIDTH
    maxX = fromInteger hEIGHT
    go (fromInteger -> y, fromInteger -> x)
      | x < 0                          = Dead
      | y < 0                          = Dead
      | x >= maxX                      = Dead
      | y >= maxY                      = Dead
      | y >= (B.length (ls !! x) - 1)  = Dead
      | B.index (ls !! x) y == '.'     = Dead
      | otherwise                      = Alive
```

The next-step value of a cell is calculate as a function of the old state of the cell and its eight adjacent cell states with the following code:

```
nextStep :: Cell -> [Cell] -> Cell
nextStep Alive adj
  | count Alive adj < 2 = Dead  -- underpopulation
  | count Alive adj > 3 = Dead  -- overpopulation
  | otherwise           = Alive  -- Alive and let Alive
nextStep Dead adj
  | count Alive adj == 3 = Alive  -- reproduction
  | otherwise            = Dead  -- nothing happens
```

The adjacent cells of a given cell are discovered via a kernel approach with the following code:

```
adjacents :: Point -> [Point]
adjacents (x,y)
  = [(x+m, y+n) | m <- [-1,0,1], n <- [-1,0,1], (m,n) /= (0,0)]
```

The `count` function referenced in the listing of `nextStep` above contains the following code:

```
count :: Eq a => a -> [a] -> Int
count x = length . filter (== x)
```

The final bit of the algorithm setup is the `gameOfLife` function:

```
gameOfLife :: Grid -> Integer -> Grid
gameOfLife initial 0 p
  = initial p
gameOfLife initial n p
  = nextStep (gameOfLife initial (n-1) p)
             (map (gameOfLife initial (n-1)) (adjacents p))
```

Note that this approach could be sped up by using memoization from `Data.MemoTrie` and `memo2` or `memoFix` but the approach above was used for this report as it more obviously predisposed itself to the parallel approach.

3.3 Sequential Approach

For the sequential approach, the main function looks as follows:

```
main :: IO ()
main = do
  [file] <- getArgs
  initialGrid <- parseInitialGrid <$> B.readFile file
  printSetup sTEP wIDTH hEIGHT
  putStrLn "finished parsing"
  let f = gameOfLife initialGrid
      calcCellAndShow point = showCell' (f sTEP point)
      calcRow row = map calcCellAndShow [(y, row) | y <- [0..(wIDTH-1)]]
  mapM_ putStrLn [calcRow x | x <- [0..(hEIGHT-1)]]
  putStrLn "done"
  printSetup sTEP wIDTH hEIGHT
```

3.4 Parallel Approach

For the parallel approach, an extremely similar approach can be used, swapping from `map` to `parMap` from the `Control.Monad`:

```
calcRow row = runPar
    (parMap calcCellAndShow [(y, row) | y <- [0..(wIDTH-1)]])
```

4 Input

4.1 Tuning Parameters

The selection of the parameters discussed above was manipulated until a reasonably tuned selection was made. The input dimensions of 1000×1000 gave a runtime of 10 to 15 seconds. Some experimentation was done with the `step` of the generation but it made for more complex parallelization as with each additional step, a whole new radius of adjacency is required for each cell calculation.

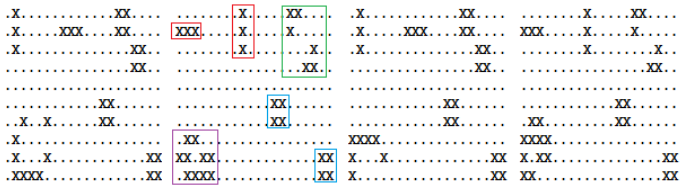
4.2 inputgen

As discussed above, the input itself does not considerably change the run-time as long as it is run to the same `step` and has the same initial grid dimensions. To test this, a variety of input files were generated using the following python script and 10 of them are included with the file listings:

```
import random

width = 1000
height = 1000

out = []

for x in range(height):
    out.append(''.join([random.choice(['X', '.']) for _ in range(width)]))

print('\n'.join(out))
```

4.3 Actual Input

Because the input does not change the runtime considerably, an input which allowed for easy visual verification of correctness of the output was selected. This was a 20×10 grid which was repeated as many times as necessary across the selected input dimensions. This grid contained two simple "blinkers" which oscillate on a two-step cycle, a slightly more complicated two-step "blinker", two stationary "blocks", and a LWSS (light weight space ship) which was oriented so as to leave the grid within a couple cycles. This input and the next three steps of it's life are shown below



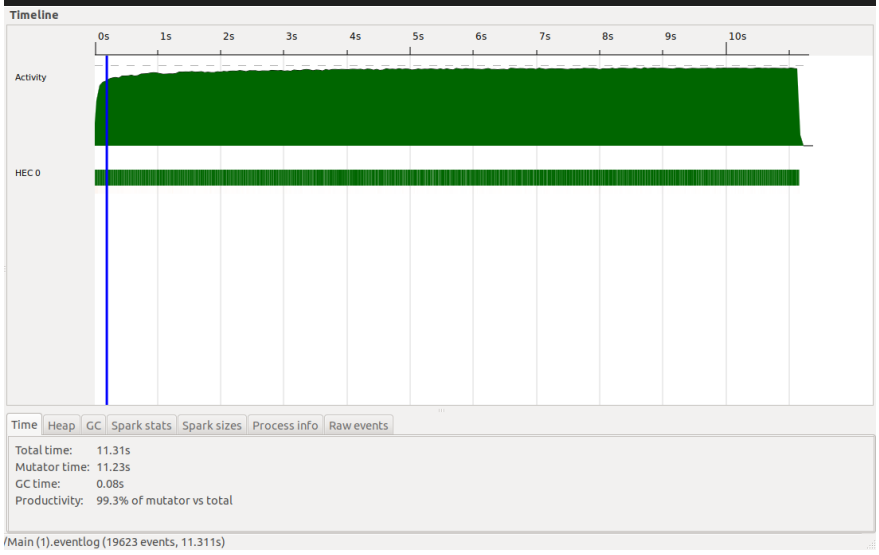
5 Results

5.1 IO Time

By running with `sTEP=0`, I found that the input-output time is around 1-1.5 seconds as opposed to the 10-12 second runtime of the `N1` algorithm for `sTEP=1`. On my machine with an AMD Ryzen 5 3600X processor with 6 cores and 12 logical cores. Below are the `-s` results and `threadscope` results for various `N` values.

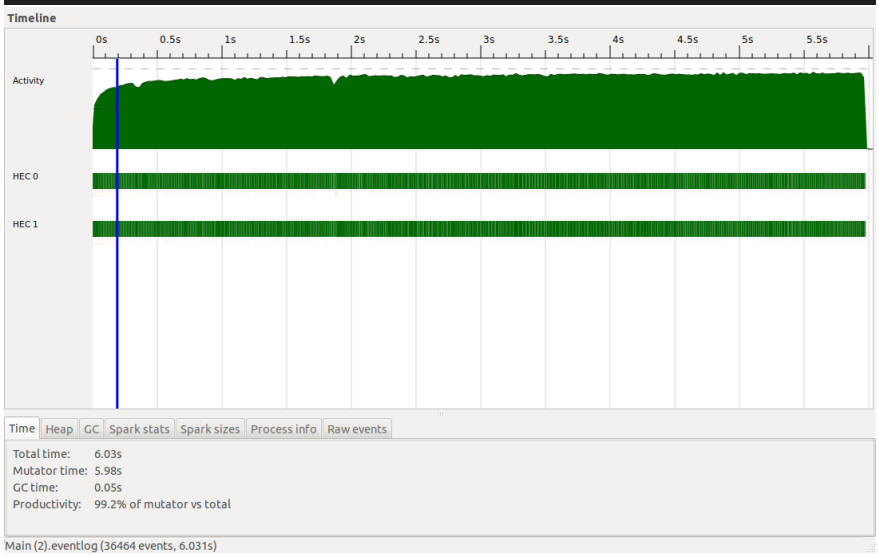
5.2 N1

```
1 1000x1000 N1
  2,284,464,456 bytes allocated in the heap
     61,936,832 bytes copied during GC
        179,312 bytes maximum residency (2 sample(s))
         29,344 bytes maximum slop
              6 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0       553 colls,     0 par    0.062s   0.079s     0.0001s    0.0004s
  Gen  1         2 colls,     0 par    0.000s   0.001s     0.0006s    0.0010s

  TASKS: 3 (1 bound, 2 peak workers (2 total), using -N1)

  SPARKS: 0 (0 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

  INIT    time    0.000s  (  0.001s elapsed)
  MUT     time   10.453s  ( 10.847s elapsed)
  GC      time    0.062s  (  0.080s elapsed)
  EXIT    time    0.000s  (  0.007s elapsed)
  Total   time   10.516s  ( 10.935s elapsed)

  Alloc rate    218,543,684 bytes per MUT second

  Productivity  99.4% of total user, 99.2% of total elapsed


real    0m11.026s
user    0m0.000s
sys     0m0.015s
```



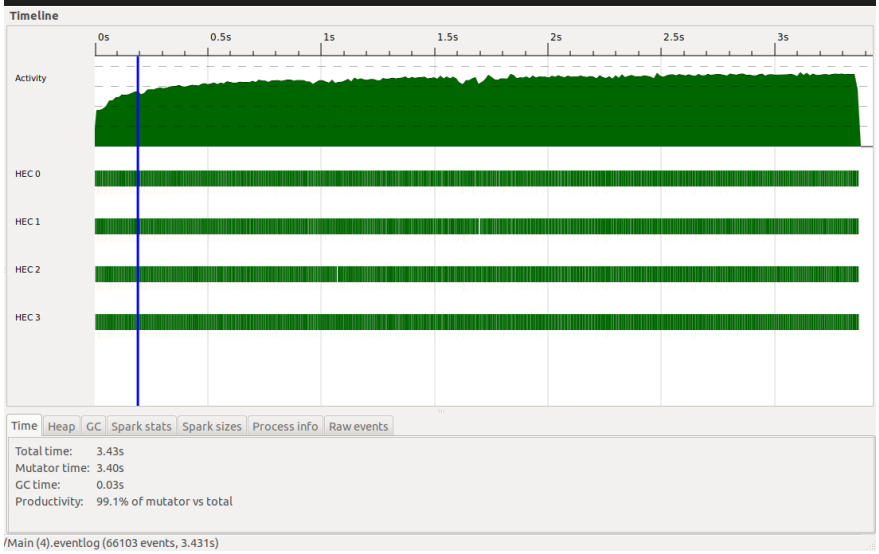
5.3 N2

```
1 1000x1000 N2
   2,315,882,904 bytes allocated in the heap
      38,291,576 bytes copied during GC
         182,224 bytes maximum residency (2 sample(s))
          37,536 bytes maximum slop
              11 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0       330 colls,   330 par    0.047s   0.051s     0.0002s    0.0004s
  Gen  1         2 colls,     1 par    0.000s   0.001s     0.0003s    0.0004s

  Parallel GC work balance: 4.43% (serial 0%, perfect 100%)

  TASKS: 4 (1 bound, 3 peak workers (3 total), using -N2)

  SPARKS: 0 (0 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

  INIT    time    0.000s  (  0.001s elapsed)
  MUT     time   10.359s  (  5.903s elapsed)
  GC      time    0.047s  (  0.052s elapsed)
  EXIT    time    0.016s  (  0.021s elapsed)
  Total   time   10.422s  (  5.977s elapsed)

  Alloc rate    223,554,307 bytes per MUT second

  Productivity  99.4% of total user, 98.8% of total elapsed


real    0m6.047s
user    0m0.000s
sys     0m0.015s
```



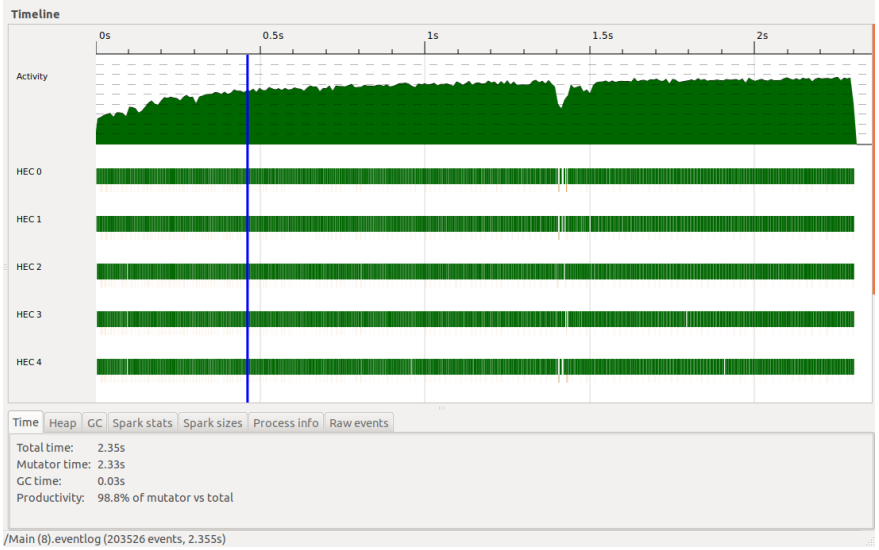
5.4 N4

```
1 1000x1000 N4
  2,386,734,784 bytes allocated in the heap
     23,382,880 bytes copied during GC
        125,888 bytes maximum residency (2 sample(s))
         58,432 bytes maximum slop
             20 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0       220 colls,   220 par    0.000s   0.037s     0.0002s    0.0004s
  Gen  1         2 colls,     1 par    0.000s   0.001s     0.0003s    0.0004s

  Parallel GC work balance: 8.91% (serial 0%, perfect 100%)

  TASKS: 6 (1 bound, 5 peak workers (5 total), using -N4)

  SPARKS: 0 (0 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

  INIT    time    0.000s  (  0.001s elapsed)
  MUT     time   11.156s  (  3.400s elapsed)
  GC      time    0.000s  (  0.037s elapsed)
  EXIT    time    0.000s  (  0.000s elapsed)
  Total   time   11.156s  (  3.438s elapsed)

  Alloc rate    213,937,011 bytes per MUT second

  Productivity 100.0% of total user, 98.9% of total elapsed


real    0m3.510s
user    0m0.000s
sys     0m0.015s
```



5.5 N8

```
1 1000x1000 N8
   2,654,842,680 bytes allocated in the heap
      18,122,520 bytes copied during GC
          44,384 bytes maximum residency (2 sample(s))
          85,160 bytes maximum slop
              37 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0       168 colls,   168 par    0.016s   0.033s     0.0002s    0.0006s
  Gen  1         2 colls,     1 par    0.000s   0.001s     0.0003s    0.0005s

  Parallel GC work balance: 12.20% (serial 0%, perfect 100%)

  TASKS: 10 (1 bound, 9 peak workers (9 total), using -N8)

  SPARKS: 0 (0 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

  INIT    time    0.000s  (  0.002s elapsed)
  MUT     time   13.812s  (  2.299s elapsed)
  GC      time    0.016s  (  0.034s elapsed)
  EXIT    time    0.000s  (  0.000s elapsed)
  Total   time   13.828s  (  2.334s elapsed)

  Alloc rate    192,205,804 bytes per MUT second

  Productivity  99.9% of total user, 98.5% of total elapsed


real    0m2.410s
user    0m0.000s
sys     0m0.015s
```

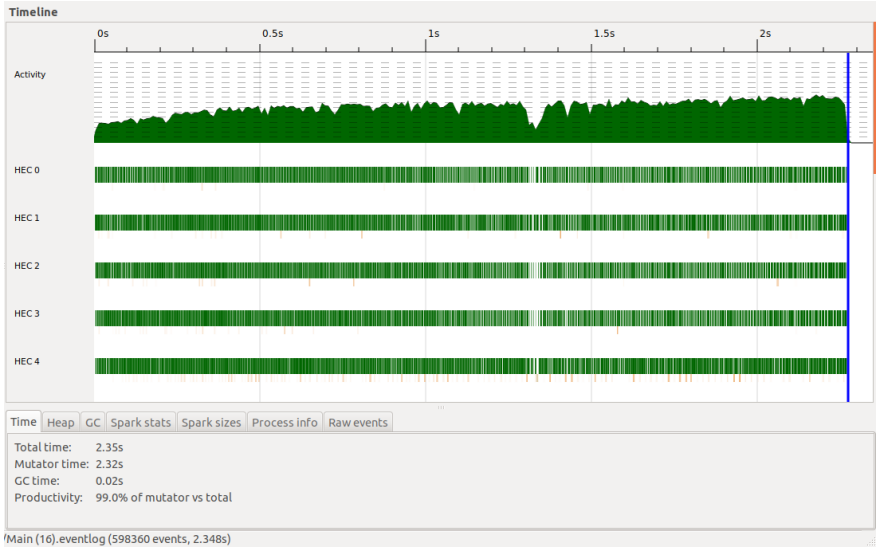


5.6 N12

```
1 1000x1000 N12
  2,776,283,208 bytes allocated in the heap
     15,073,448 bytes copied during GC
        131,472 bytes maximum residency (2 sample(s))
        126,576 bytes maximum slop
             53 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0       155 colls,   155 par    0.000s   0.027s     0.0002s    0.0004s
  Gen  1         2 colls,     1 par    0.000s   0.001s     0.0004s    0.0005s

  Parallel GC work balance: 17.03% (serial 0%, perfect 100%)

  TASKS: 14 (1 bound, 13 peak workers (13 total), using -N12)

  SPARKS: 0 (0 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

  INIT    time    0.000s  (  0.002s elapsed)
  MUT     time   14.516s  (  2.139s elapsed)
  GC      time    0.000s  (  0.028s elapsed)
  EXIT    time    0.000s  (  0.000s elapsed)
  Total   time   14.516s  (  2.169s elapsed)

  Alloc rate    191,261,706 bytes per MUT second

  Productivity 100.0% of total user, 98.6% of total elapsed


real    0m2.248s
user    0m0.000s
sys     0m0.015s
```

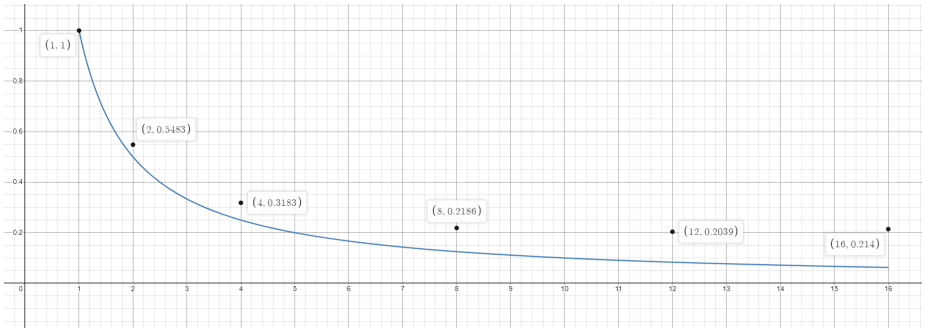
5.7 N16

```
1 1000x1000 N16
   3,086,019,904 bytes allocated in the heap
      16,489,680 bytes copied during GC
         102,984 bytes maximum residency (2 sample(s))
         152,224 bytes maximum slop
              70 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0       159 colls,   159 par    0.047s   0.032s     0.0002s    0.0006s
  Gen  1         2 colls,     1 par    0.000s   0.001s     0.0004s    0.0006s

  Parallel GC work balance: 21.17% (serial 0%, perfect 100%)

  TASKS: 18 (1 bound, 17 peak workers (17 total), using -N16)

  SPARKS: 0 (0 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

  INIT    time    0.000s  (  0.003s elapsed)
  MUT     time   15.250s  (  2.250s elapsed)
  GC      time    0.047s  (  0.033s elapsed)
  EXIT    time    0.000s  (  0.000s elapsed)
  Total   time   15.297s  (  2.286s elapsed)

  Alloc rate    202,361,960 bytes per MUT second

  Productivity  99.7% of total user, 98.4% of total elapsed


real    0m2.360s
user    0m0.000s
sys     0m0.015s
```



5.8 Speedup

Below is a graph of the speedup as a percentage of ideal speedup using the timings reported in the `N_` sections above. The x axis is the number of cores and the y access is the runtime relative to the `N1` runtime.



6 Conclusion

From the results above we can see that speedup is achieved at lower core counts but falls off with higher core counts. We can see from the threadscope results that thread utilization is quite high but there is more and more gaps (possibly due to GC) at the higher thread counts. This is likely due to the amount of duplicated calls between the threads due to the `adjacents` call in the logic. This could be made faster through better slicing of the input and memoization. The largest drawback of the approach is the original data representation adapted from the blog mentioned originally as well as the initial grid parsing which it employs. It makes for large inefficiency and unnecessary random access over the data. By employing a structure which could more easily be traversed and zipped over without the unnecessary access the original algorithm could be made much faster. Re-evaluating the data organization as well as the `parseInitialGrid` could help to achieve this goal.