

Cellular Automaton (Life2) - Final Report

Eric Jing (epj2117)

Introduction

Cellular automata are a family of curious algorithms that have sparked interest in various fields of computing and mathematics. Their ability to produce complex and unpredictable states from comparatively simple rules makes them one of the best examples of emergent phenomena - the observation that simple rules can give rise to phenomena much more intricate than their simplicity would suggest.

One algorithm in particular has gained disproportionate notoriety: Conway's Game of Life. John Conway proposed a set of rules that are analogous to real world biology. From those rules spring forth endless permutations of the state grid, with various creations featured on many odd corners of the internet. The algorithm is even Turing complete: it can simulate a computer with its rules, including the computer that is currently running it.

Algorithm

Given a 2D grid of cells that could either be "dead" or "alive":

1. If a live cell has fewer than two neighbors, it dies the next turn, as if by loneliness.
2. If a live cell has more than three neighbors, it dies the next turn, as if by overcrowding.
3. If a dead cell has exactly three neighbors, it comes alive the next turn, as if through reproduction.

There are many ways to handle edge cases. The way this implementation handles it is to have the edges wrap around to the other side. Topologically, the grid is equivalent to a torus.

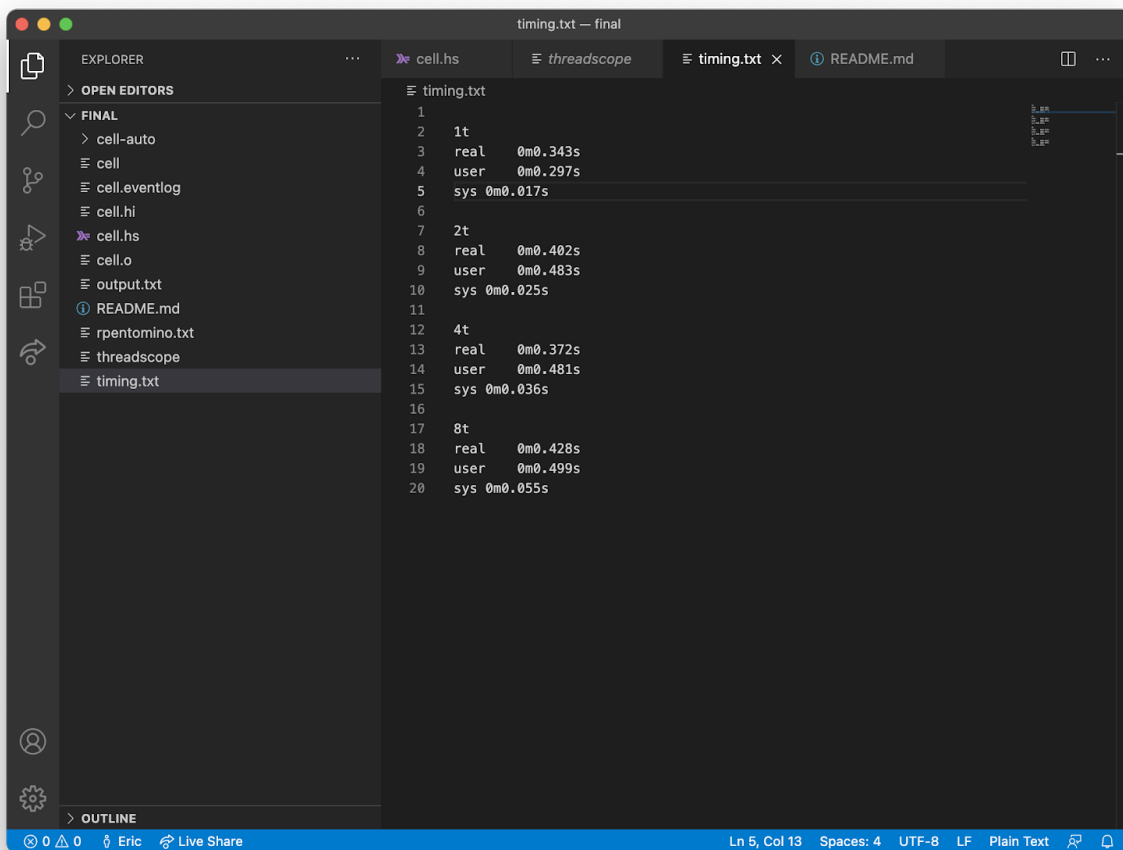
Haskell Implementation

To implement a parallel simulator in Haskell, the most obvious way to parallelize the algorithm is to break the 2D grid up into smaller subgrids. These subgrids can then be operated upon in parallel. The rules describe local behavior, and can easily be applied to every cell in a grid. The first caveat is handling the edge cases. Cells at the edges of a subgrid would need to know about neighbors in another subgrid. Therefore, every subgrid must be queried for its edges, in order to provide its neighbors the necessary information to compute their edge cells.

To do this, each subgrid is represented by a vector of Int data types that can be 1 or 0, along with metadata for rows and columns. These subgrids are stored in another vector, representing the entire 2D grid. Every time the board is updated, new subgrids are created with values obtained from previously-calculated subgrids, as well as the edges of their neighbors.

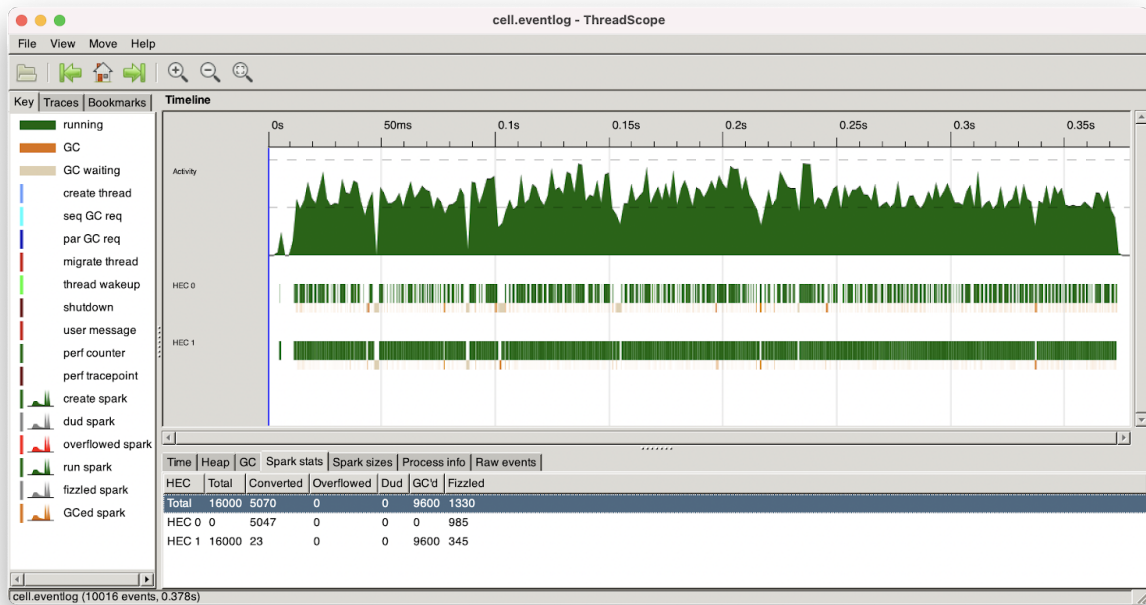
Parallelization is done through the use of Haskell constructs such as `parTraversable` to evaluate the vector of subgrids in parallel. Every iteration, the main Haskell thread will create sparks for every subgrid. Once the entire board is updated, another round of sparks will be generated to compute the next updated state.

Results

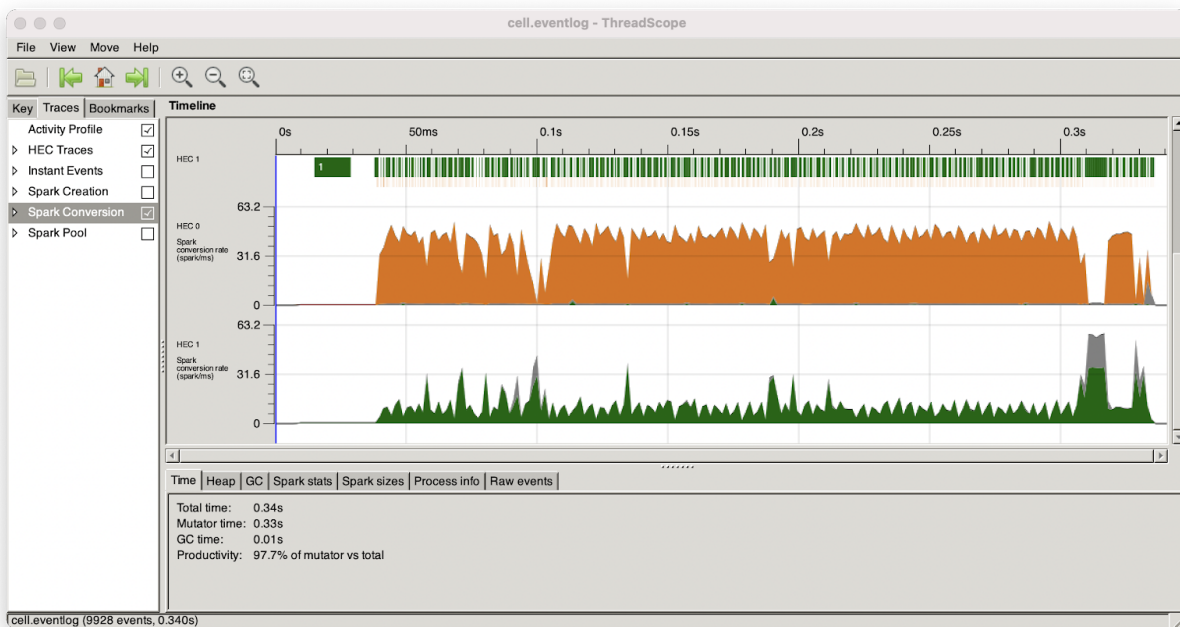


```
timing.txt
1
2 1t
3 real 0m0.343s
4 user 0m0.297s
5 sys 0m0.017s
6
7 2t
8 real 0m0.402s
9 user 0m0.483s
10 sys 0m0.025s
11
12 4t
13 real 0m0.372s
14 user 0m0.481s
15 sys 0m0.036s
16
17 8t
18 real 0m0.428s
19 user 0m0.499s
20 sys 0m0.055s
```

The program was tested on a Macbook Pro, with a dual-core Intel i5 processor. The test input was a 64*64 board of an r-pentomino, simulated 1000 times.



While the program produced the correct result in every case, there was no significant speed-up observed.



Looking at the ThreadScope measurements, almost all of the created sparks were garbage-collected. In hindsight, this was normal. As only the final iteration was necessary for the output, there is no need to keep the other 999 results. Because the main thread creates the sparks for the rest of the threads, it performs the bulk of the work, while the other threads do what they can with the work made available.

Several things could be causing this result. One of them, the hardware, was the most likely culprit. If the two-thread test underperformed, it was easy to see why adding more threads would not help improve the performance. Another cause could be that the program was not as taxing on the processor as it is on the memory. Since each computation is essentially a lookup of a cell and its neighbors, memory that is accessed has relatively few computations performed upon it. If memory is indeed the bottleneck, then no amount of threading or sparking can overcome the performance gap.

Conclusion

While Haskell can be very performant under the right circumstances, given the simple nature of the computation performed it falls short of the anticipated performance gains that this algorithm suggests. A potential solution, aside from upgrading hardware, would be to optimize the algorithm further. Using knowledge of hardware features such as caching, the algorithm can be moved to a more high-performance language (like C++), where the lack of a garbage collector and manual memory management grants a massive performance boost. Each subgrid can be positioned in memory so that the processor always has complete subgrids in the same are of cache, to reduce main memory lookup to a minimum.

Program Listing

cell.hs

```
import qualified Data.Vector as BV
import qualified Data.Vector.Mutable as BMV

import qualified Data.Vector.Unboxed as V
import qualified Data.Vector.Unboxed.Mutable as MV
import Data.Bits
import Control.Monad.ST
import Control.DeepSeq
import qualified Control.Monad.Primitive as P

import Data.Char
import qualified System.Environment
import qualified System.Exit
import System.IO
import Control.Parallel
import Control.Parallel.Strategies
import Control.Monad (forM, forM_, guard)

data Grid = Grid {
  gData :: V.Vector Int,
  gRows :: Int,
  gCols :: Int
}
instance NFData Grid where
  rnf g = rnf (gData g) `seq` rnf (gRows g) `seq` rnf (gCols g)

data Edge = Edge {
  eUL :: Int,
  eUU :: [Int],
  eUR :: Int,
  eLL :: [Int],
  eRR :: [Int],
  eDL :: Int,
  eDD :: [Int],
  eDR :: Int
}

-- Gets edge info of grid
edgeOfGrid :: Grid -> Edge
```

```

edgeOfGrid g = Edge {
  eUL = gData g V.! coordsToIndex (gCols g) (0,0),

  eUU = [gData g V.! coordsToIndex (gCols g) (0,i)
    | i <- [0..(gCols g - 1)]],

  eUR = gData g V.! coordsToIndex (gCols g) (0,gCols g - 1),

  eLL = [gData g V.! coordsToIndex (gCols g) (i,0)
    | i <- [0..(gRows g - 1)]],

  eRR = [gData g V.! coordsToIndex (gCols g) (i,gCols g - 1)
    | i <- [0..(gRows g - 1)]],

  eDL = gData g V.! coordsToIndex (gCols g) (gRows g - 1,0),

  eDD = [gData g V.! coordsToIndex (gCols g) (gRows g - 1,i)
    | i <- [0..(gCols g - 1)]],

  eDR = gData g V.! coordsToIndex (gCols g) (gRows g - 1,gCols g - 1)
}

```

```

coordsToIndex :: Int -> (Int,Int) -> Int
coordsToIndex cols (r,c) = r*cols+c

```

```

coordRange :: [Int] -> [Int] -> [(Int,Int)]
coordRange rs cs = do
  r <- rs
  c <- cs
  return (r, c)

```

-- Simulates a grid, plus external edges

```

nextStep :: Grid -> Edge -> Grid
nextStep g e = g {gData = V.create nvec}
  where
    nvec :: ST s (MV.MVector s Int)
    nvec = do
      v <- MV.replicate (V.length $ gData g) 0
      --Handle corners
      MV.write v (coordsToIndex (gCols g) (0,0))          (eUL e)
      MV.write v (coordsToIndex (gCols g) (0,gCols g - 1)) (eUR e)
      MV.write v (coordsToIndex (gCols g) (gRows g - 1,0)) (eDL e)
      MV.write v (coordsToIndex (gCols g) (gRows g - 1,gCols g - 1)) (eDR e)
      --Handle edges

```

```

mapM_ \(i,elem) -> MV.modify v
    (+elem)
    (coordsToIndex (gCols g) (0, i)))
(zip [0..gCols g - 1] (eUU e))
mapM_ \(i,elem) -> MV.modify v
    (+elem)
    (coordsToIndex (gCols g) (0, i)))
(zip [1..gCols g - 1] (eUU e))
mapM_ \(i,elem) -> MV.modify v
    (+elem)
    (coordsToIndex (gCols g) (0, i)))
(zip [0..gCols g - 2] (tail $ eUU e))

mapM_ \(i,elem) -> MV.modify v
    (+elem)
    (coordsToIndex (gCols g) (gRows g - 1, i)))
(zip [0..gCols g - 1] (eDD e))
mapM_ \(i,elem) -> MV.modify v
    (+elem)
    (coordsToIndex (gCols g) (gRows g - 1, i)))
(zip [1..gCols g - 1] (eDD e))
mapM_ \(i,elem) -> MV.modify v
    (+elem)
    (coordsToIndex (gCols g) (gRows g - 1, i)))
(zip [0..gCols g - 2] (tail $ eDD e))

mapM_ \(i,elem) -> MV.modify v
    (+elem)
    (coordsToIndex (gCols g) (i, 0)))
(zip [0..gRows g - 1] (eLL e))
mapM_ \(i,elem) -> MV.modify v
    (+elem)
    (coordsToIndex (gCols g) (i, 0)))
(zip [1..gRows g - 1] (eLL e))
mapM_ \(i,elem) -> MV.modify v
    (+elem)
    (coordsToIndex (gCols g) (i, 0)))
(zip [0..gRows g - 2] (tail $ eLL e))

mapM_ \(i,elem) -> MV.modify v
    (+elem)
    (coordsToIndex (gCols g) (i, gCols g - 1)))
(zip [0..gRows g - 1] (eRR e))

```

```

mapM_ \(i,elem) -> MV.modify v
    (+elem)
    (coordsToIndex (gCols g) (i, gCols g - 1)))
(zip [1..gRows g - 1] (eRR e))
mapM_ \(i,elem) -> MV.modify v
    (+elem)
    (coordsToIndex (gCols g) (i, gCols g - 1)))
(zip [0..gRows g - 2] (tail $ eRR e))
--Look at neighboring cells from all 8 dirs
mapM_ \(r,c) -> MV.modify v
    (+gData g V.! (coordsToIndex (gCols g) (r, c)))
    (coordsToIndex (gCols g) (r+1, c+1) ))
    (coordRange [0..(gRows g - 2)] [0..(gCols g - 2)])
mapM_ \(r,c) -> MV.modify v
    (+gData g V.! (coordsToIndex (gCols g) (r, c)))
    (coordsToIndex (gCols g) (r, c+1) ))
    (coordRange [0..(gRows g - 1)] [0..(gCols g - 2)])
mapM_ \(r,c) -> MV.modify v
    (+gData g V.! (coordsToIndex (gCols g) (r, c)))
    (coordsToIndex (gCols g) (r-1, c+1) ))
    (coordRange [1..(gRows g - 1)] [0..(gCols g - 2)])
mapM_ \(r,c) -> MV.modify v
    (+gData g V.! (coordsToIndex (gCols g) (r, c)))
    (coordsToIndex (gCols g) (r+1, c) ))
    (coordRange [0..(gRows g - 2)] [0..(gCols g - 1)])
mapM_ \(r,c) -> MV.modify v
    (+gData g V.! (coordsToIndex (gCols g) (r, c)))
    (coordsToIndex (gCols g) (r-1, c) ))
    (coordRange [1..(gRows g - 1)] [0..(gCols g - 1)])
mapM_ \(r,c) -> MV.modify v
    (+gData g V.! (coordsToIndex (gCols g) (r, c)))
    (coordsToIndex (gCols g) (r+1, c-1) ))
    (coordRange [0..(gRows g - 2)] [1..(gCols g - 1)])
mapM_ \(r,c) -> MV.modify v
    (+gData g V.! (coordsToIndex (gCols g) (r, c)))
    (coordsToIndex (gCols g) (r, c-1) ))
    (coordRange [0..(gRows g - 1)] [1..(gCols g - 1)])
mapM_ \(r,c) -> MV.modify v
    (+gData g V.! (coordsToIndex (gCols g) (r, c)))
    (coordsToIndex (gCols g) (r-1, c-1) ))
    (coordRange [1..(gRows g - 1)] [1..(gCols g - 1)])

--Calculate final cell state from sum of neighbors
mapM_ \(i -> MV.modify v

```



```

        (\s -> ((1-(gData g V.! i))
            .&. fromEnum (s == 3))
            .|. ((gData g V.! i)
            .&. fromEnum (s `shift` (-1) == 1)))
        i)
    [0..(V.length $ gData g) - 1]

```

```

return v

```

```

readGrid :: Handle -> IO Grid

```

```

readGrid h = do

```

```

    [rowstr,colstr] <- words <$> hGetLine h

```

```

    rows <- readIO rowstr

```

```

    cols <- readIO colstr

```

```

    cells <- map digitToInt

```

```

        <$> filter (\c -> c=='0' || c=='1')

```

```

        <$> hGetContents h

```

```

    return Grid{

```

```

        gRows = rows,

```

```

        gCols = cols,

```

```

        gData = V.fromList cells

```

```

    }

```

```

data MultiGrid = MultiGrid {

```

```

    mGrids :: BV.Vector Grid,

```

```

    mRows :: Int,

```

```

    mCols :: Int,

```

```

    mSubRows :: Int,

```

```

    mSubCols :: Int

```

```

}

```

```

instance NFData MultiGrid where

```

```

    rnf mg = rnf (mGrids mg) `seq`

```

```

        rnf (mRows mg) `seq`

```

```

        rnf (mCols mg) `seq`

```

```

        rnf (mSubRows mg) `seq`

```

```

        rnf (mSubCols mg)

```

```

subGridOffset x m si = x*si `div` m

```

```

coordToSubGrid x m i = (m-1)-((x-1-i)*m `div` x)

```

```

getCell :: MultiGrid -> (Int,Int) -> Int

```

```

getCell mg (r,c) = gData subg V.!

```

```

    coordsToIndex

```

```

    (gCols subg)

```

```

(r-subGridOffset (mRows mg) (mSubRows mg) rd,
 c-subGridOffset (mCols mg) (mSubCols mg) cd)
where rd = coordToSubGrid (mRows mg) (mSubRows mg) r
      cd = coordToSubGrid (mCols mg) (mSubCols mg) c
      subg = mGrids mg BV.!
          coordsToIndex (mSubCols mg) (rd, cd)

```

```

makeSubGrid :: Grid -> (Int,Int) -> (Int,Int) -> Grid
makeSubGrid g (rows, cols) (ro, co) = Grid{
  gData = V.generate (rows*cols)
    (\i -> let (ri,ci) = divMod i cols in
            gData g V.! coordsToIndex
              (gCols g)
              (ro + ri, co + ci)),
  gRows = rows,
  gCols = cols
}

```

```

subdivideGrid :: (Int,Int) -> Grid -> MultiGrid
subdivideGrid (mr,mc) g = MultiGrid{
  mGrids = BV.fromList $
    map (\(sr,sc) -> makeSubGrid g
      (sgor (sr+1) - sgor sr, sgoc (sc+1) - sgoc sc)
      (sgor sr,sgoc sc))
      (coordRange [0..mr-1] [0..mc-1]),

```

```

  mRows = gRows g,
  mCols = gCols g,
  mSubRows = mr,
  mSubCols = mc
}

```

```

where sgor = subGridOffset (gRows g) mr
      sgoc = subGridOffset (gCols g) mc
--Calculates one subgrid of new multigrid
nextStepMG :: MultiGrid -> (Int,Int) -> Grid
nextStepMG mg (r, c) = nextStep grid edge
where grid = mGrids mg BV.! coordsToIndex (mSubCols mg) (r,c)
      edge = Edge{
        eUL = eDR $ edgeOfGrid $ (mGrids mg) BV.!
          coordsToIndex (mSubCols mg) (
            (r-1) `mod` mSubRows mg,

```

```

        (c-1)`mod` mSubCols mg
    ),
    eUU = eDD $ edgeOfGrid $ (mGrids mg) BV.!
        coordsToIndex (mSubCols mg) (
            (r-1)`mod` mSubRows mg,
            (c)`mod` mSubCols mg
        ),
    eUR = eDL $ edgeOfGrid $ (mGrids mg) BV.!
        coordsToIndex (mSubCols mg) (
            (r-1)`mod` mSubRows mg,
            (c+1)`mod` mSubCols mg
        ),
    eLL = eRR $ edgeOfGrid $ (mGrids mg) BV.!
        coordsToIndex (mSubCols mg) (
            (r)`mod` mSubRows mg,
            (c-1)`mod` mSubCols mg
        ),
    eRR = eLL $ edgeOfGrid $ (mGrids mg) BV.!
        coordsToIndex (mSubCols mg) (
            (r)`mod` mSubRows mg,
            (c+1)`mod` mSubCols mg
        ),
    eDL = eUR $ edgeOfGrid $ (mGrids mg) BV.!
        coordsToIndex (mSubCols mg) (
            (r+1)`mod` mSubRows mg,
            (c-1)`mod` mSubCols mg
        ),
    eDD = eUU $ edgeOfGrid $ (mGrids mg) BV.!
        coordsToIndex (mSubCols mg) (
            (r+1)`mod` mSubRows mg,
            (c)`mod` mSubCols mg
        ),
    eDR = eUL $ edgeOfGrid $ (mGrids mg) BV.!
        coordsToIndex (mSubCols mg) (
            (r+1)`mod` mSubRows mg,
            (c+1)`mod` mSubCols mg
        )
}

```

nextMultiGridV :: MultiGrid -> BV.Vector Grid

```

nextMultiGridV mg = runEval $ parTraversable rdeepseq $ BV.imap
    (\i g -> nextStepMG mg (i `divMod` mSubCols mg))

```

```
(mGrids mg)
```

```
simulate :: Int -> MultiGrid -> MultiGrid
simulate 0 mg = mg
simulate i mg = nv `deepseq` simulate (i-1) newg
  where nv = nextMultiGridV mg
        newg = mg {mGrids = nv}
```

```
main :: IO ()
main = do
  args <- System.Environment.getArgs
  if (length args /= 1)
  then do
    System.Exit.die "Usage: cell <iterations>"
  else do
```

```
  iters <- readIO $ head args :: IO Int
  gridinit <- readGrid stdin
  let multigrid = subdivideGrid (r,c) gridinit
      where r = gRows gridinit `div` 16
            c = gCols gridinit `div` 16
```

```
  let mg = simulate iters multigrid
```

```
  hPutStr stdout $ show $ mRows mg
  hPutChar stdout ' '
  hPutStrLn stdout $ show $ mCols mg
  forM [0..mRows mg - 1] $ \r -> do
    forM [0..mCols mg - 1] $ \c -> do
      hPutChar stdout (intToDigit $ getCell mg (r,c))
    hPutChar stdout '\n'
  return ()
```