COMS 4995 Final Project

Zehua Chen (zc2616)

December 22, 2021

Contents

1 Implementation

1.1 Coordinate System

The world has a height and a width. The origin (0, 0) is located at the center of the world. If we save the world as an image, positive axis would be on the right of the image and positive y axis would be on the top of the image.

In order for a cell to exist at (0, 0) and the world be modeled using width and height, width and height must be odd numbers

1.2 Single Core

Single core implementation (`Conway.Simulate.simulateSync`) of the simulation:

1. Simulate the world
2. Simulate one layer outside of the existing world, and expand the existing world if needed

1.3 Multi Core

Multicore implementation (`Conway.Simulate.simulateAsync`) of the simulation

1. Divide the world into partitions and calculate adjacent cells between partitions
2. Simulate the partitions, with access only to the world within the partition. *This step reuses code of step 1 from single core implementation*
3. Simulate the adjacent cells between partitions, with access to the whole world.
4. Simulate one layer outside of the existing world. *This step reuses code of step 1 from single core implementation*
5. Combine the result of 2, 3 and 4

Of the above steps, the operations are 1 is performed in parallel first. After they finish, the operations in 2, 3, 4 are performed in parallel. The latter operations are performed in a second step because they depend on the result from the former operations.

1.3.1 Parameters

Multicore simulation allows customizations in

- slice width, slice height: how big each slice should be
- chunk size: the multi core simulation uses `parList` in some operations. Chunk size is used to customize `parList`

1.4 Modules

1. `Conway.World`: provide abstraction to the conway world, and various utility functions useful for operations on the world
2. `Conway.Slice`: provide abstraction of slices of worlds
3. `Conway.Partition`: provide function that split the world into slices, and find the cells on the outer layer of each partition
4. `Conway.Simulate`: implements single and multi core simulation
5. `Conway.PPM`: saves a conway world into a PPM image

1.5 Testing

1. `test/World.hs`: tests for `Conway.World`
2. `test/Partition/Partition.hs`: make sure the world can be divided into partitions
3. `test/Partition/PartitionBorders.hs`: make sure the outer most layer of a partition can be resolved property
4. `test/Simulate/Finite.hs`: make sure that simulation on finite grid is correct
5. `test/Simulate/Grow.hs`: make sure that teh grid is grown when needed
6. `test/Simulate/Infinite.hs`: make sure that single core and multi core implementation produces the same result from simulating on an infinite grid.

1.6 Command Line Interface

1.6.1 Simulation

In order to run simulation, pass the following to the command line application

```
conway-exe <file> <iterations> <slice width> <slice height> <chunk size>
```

"file" is a json file containing the description of the world. The file should look like the following.

```
{
  "width": 4,
  "height": 4,
  "livingCells": [
    { "x": 0, "y": 0 },
    { "x": 0, "y": 1 },
    { "x": 0, "y": 2 }
  ]
}
```

After the simulation has finished, the program will save two images `sync.ppm` and `async.ppm`, one produced by single core simulation, and the other produced by multi core simulation.

1.6.2 Benchmarking

In order to benchmark, give the following to the command line application

```
conway-exe benchmark
```

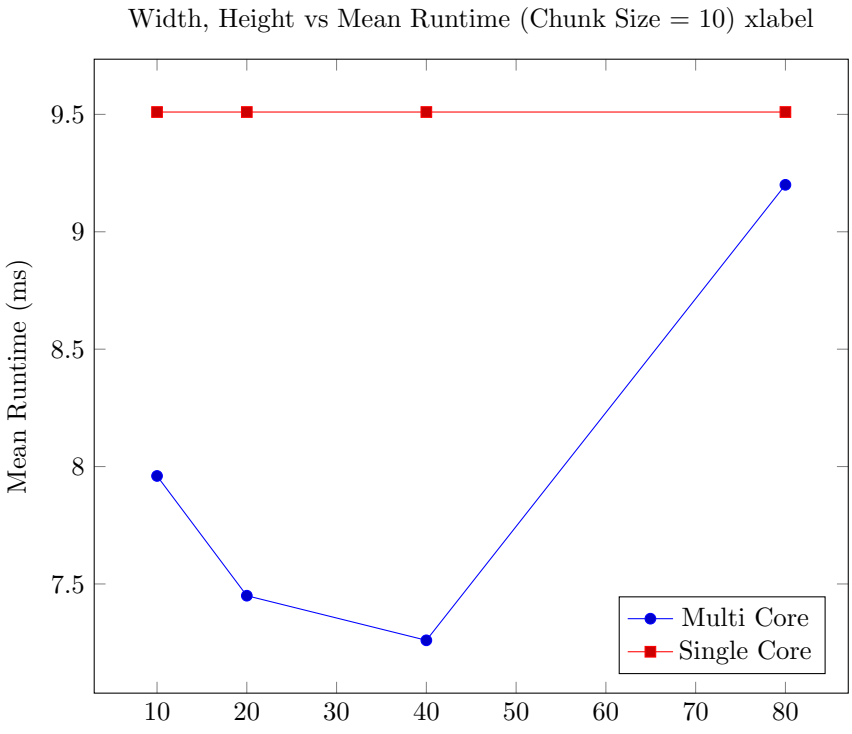
2 Performance

My machine has the following specs

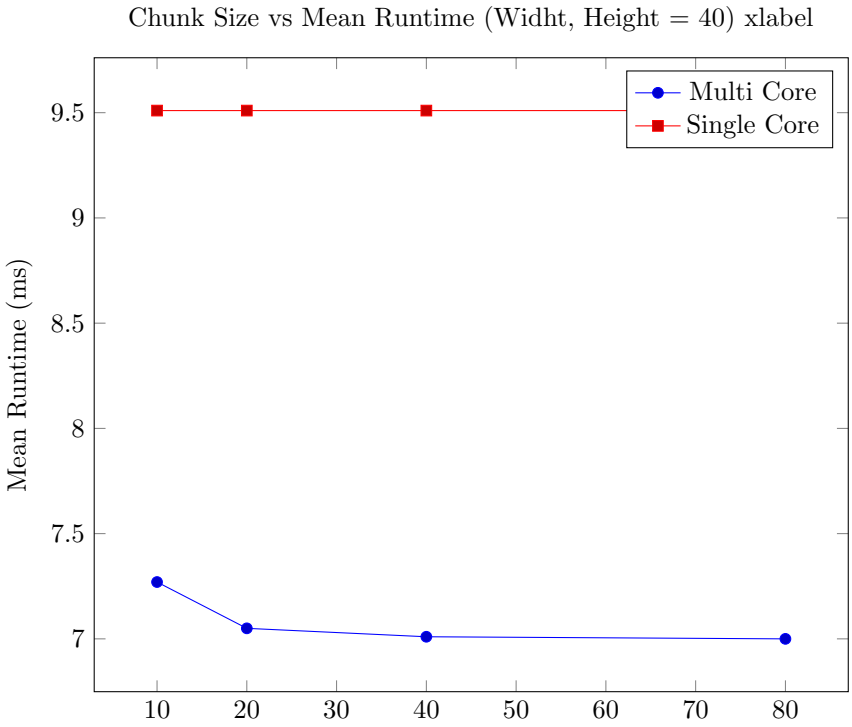
- 2 energy efficient cores
- 6 performance cores
- 16 GB of memory

Benchmarking is done using `Criterion.Main`

2.1 Single Iteration on World of Size 101 x 101

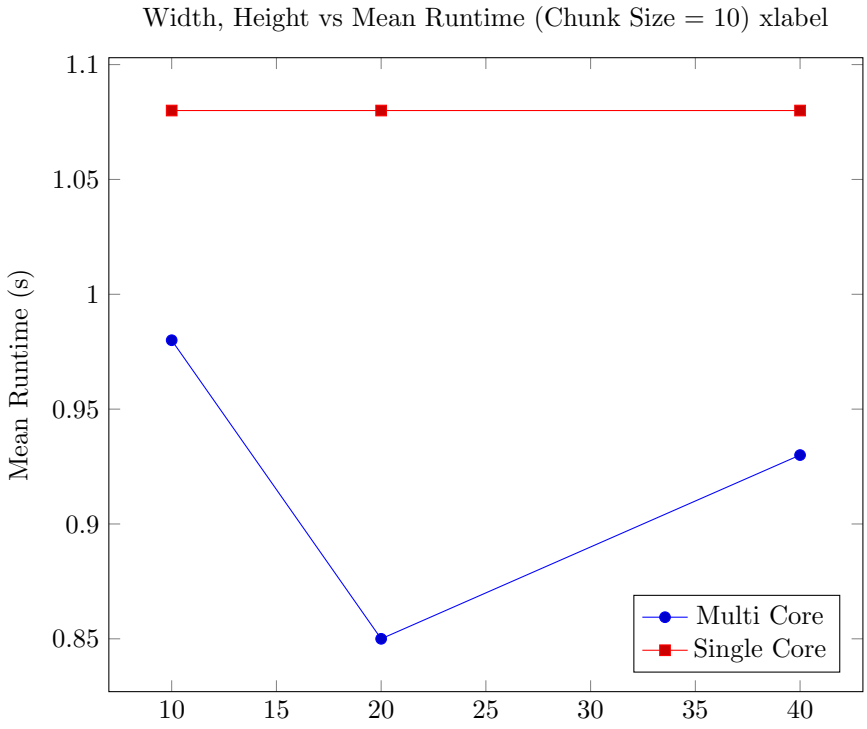


The above data is generated by changing the slice width and slice height of async simulation, while keeping chunk size at 10.

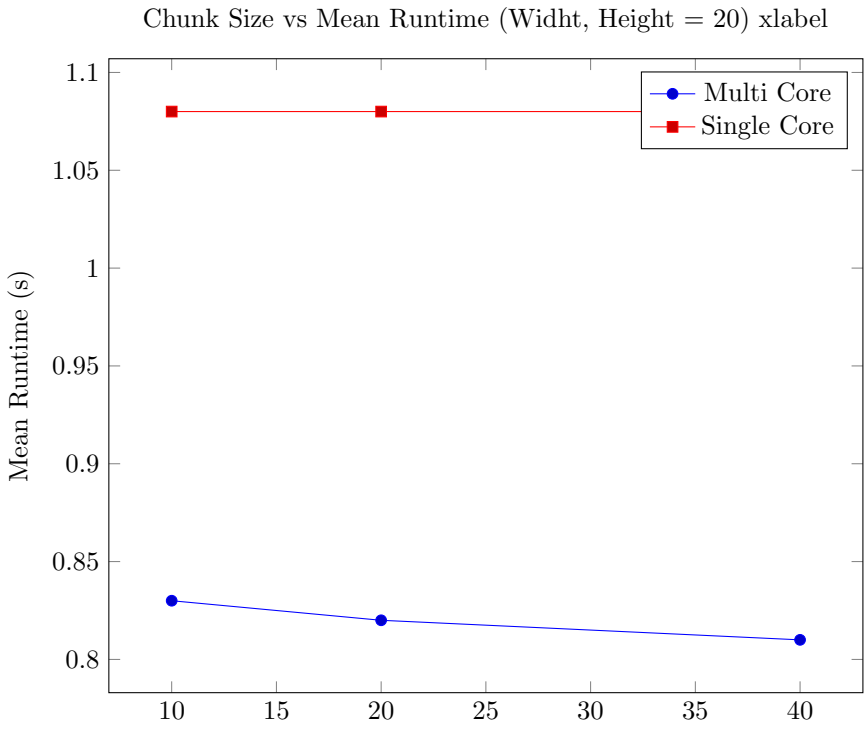


The above data is generated by changing the chunk size while keeping slice width and slice height at 40

2.2 100 Iterations on World of Size 51 vs 51



The above data is generated by changing the slice width and slice height of async simulation, while keeping chunk size at 10.



The above data is generated by changing the chunk size while keeping slice width and slice height at 20

2.3 Conclusion

From the above performance figures, the following can be concluded

- The multi core implementation takes around 80% of the time it takes for single core implementations regardless of the number of iterations and world size
- Increasing chunk size can reduce performance
- Slice width and height needs tuning in order to achieve the best performance. On my machine the best slice width and height should be around half of the world width and height.

3 Code

3.1 Command Line Application

Listing 1: `Main.hs`

```
1  module Main where
2
3  import qualified Benchmark
4  import qualified Conway.Json as Json
5  import qualified Conway.PPM as PPM
6  import qualified Conway.Simulate as Simulate
7  import System.Environment
8
9  program :: [String] -> IO ()
```

```
program ["benchmark"] = do
  Benchmark.benchmark
program [input, iterations, sliceWidth, sliceHeight, chunkSize] = do
  start <- Json.worldFromJson input
  -- https://stackoverflow.com/questions/20667478/haskell-string-int-type-conversion
  let iterations' = read iterations :: Int
      sliceWidth' = read sliceWidth :: Int
      sliceHeight' = read sliceHeight :: Int
      chunkSize' = read chunkSize :: Int
      sync = foldr (\_ w -> Simulate.simulateSync w) start [0 .. iterations']
      async = foldr (\_ w -> Simulate.simulateAsync sliceWidth' sliceHeight' chunkSize' w) start [0 .. iterations

  PPM.save sync "sync.ppm"
  PPM.save async "async.ppm"
program args = do
  putStr $
    "args not recognized: " ++ show args ++ "\n"
      ++ "zc2616's final project\n"
      ++ "  benchmark: run bench mark\n"
      ++ "  <file> <iterations> <slice width> <slice height> <chunk size>: simulate file\n"

main :: IO ()
main = do
  args <- getArgs
  program args
```

Listing 2: Benchmark.hs

```
module Benchmark (benchmark) where

import qualified Conway.Simulate as Simulate
import qualified Conway.World as World
import Criterion.Main

bigWorld :: World.World
bigWorld = World.fromList (replicate 101 (replicate 101 True))

mediumWorld :: World.World
mediumWorld = World.fromList (replicate 51 (replicate 51 True))

simulate :: Int -> (World.World -> World.World) -> (World.World -> World.World)
simulate count f start = foldr (\_ w -> f w) start [1 .. count]

benchSingleIteration :: Benchmark
benchSingleIteration =
  bgroup
    "single iteration, world of size 101 x 101"
    [ bench "sync" sync,
      bench "w: 10, h: 10, chunk: 10" (async 10 10 10),
      bench "w: 20, h: 20, chunk: 10" (async 20 20 10),
      bench "w: 40, h: 40, chunk: 10" (async 40 40 10),
      bench "w: 80, h: 80, chunk: 10" (async 80 80 10),
      bench "w: 40, h: 40, chunk: 10" (async 40 40 10),
      bench "w: 40, h: 40, chunk: 20" (async 40 40 20),
      bench "w: 40, h: 40, chunk: 40" (async 40 40 40),
      bench "w: 40, h: 40, chunk: 80" (async 40 40 80)
    ]
  where
    sync = nf (simulate 1 Simulate.simulateSync) bigWorld
    async w h c = nf (simulate 1 (Simulate.simulateAsync w h c)) bigWorld

benchMultipleIterations :: Benchmark
benchMultipleIterations =
  bgroup
    "100 iterations, world of size 51 x 51"
    [ bench "sync" sync,
      bench "w: 10, h: 10, chunk: 10" (async 10 10 10),
      bench "w: 20, h: 20, chunk: 10" (async 20 20 10),
```

```
41        bench "w: 40, h: 40, chunk: 10" (async 40 40 10),
42        bench "w: 20, h: 20, chunk: 10" (async 20 20 10),
43        bench "w: 20, h: 20, chunk: 20" (async 20 20 20),
44        bench "w: 20, h: 20, chunk: 40" (async 20 20 40)
45      ]
46    where
47      sync = nf (simulate 100 Simulate.simulateSync) mediumWorld
48      async w h c = nf (simulate 100 (Simulate.simulateAsync w h c)) mediumWorld
49
50  benchmark :: IO ()
51  benchmark =
52    defaultMain
53      [ benchSingleIteration,
54        benchMultipleIterations
55      ]
```

3.2 Implementation

Listing 3: Conway.Partition

```
1   {-# LANGUAGE TupleSections #-}
2
3   module Conway.Partition
4     ( Slice (Slice, minX, maxX, minY, maxY),
5       partition,
6       partitionBorders,
7       fromWorld,
8     )
9   where
10
11  import Conway.Slice
12  import qualified Conway.World as World
13  import qualified Data.HashSet as Set
14
15  partition :: Int -> Int -> World.World -> [Slice]
16  partition sliceWidth sliceHeight world =
17    map
18      ( \((xMin, xMax), (yMin, yMax)) ->
19          Slice {minX = xMin, maxX = xMax, minY = yMin, maxY = yMax}
20      )
21      xys
22    where
23      slice :: Int -> Int -> Int -> [World.Vec2] -> [World.Vec2]
24      slice current end step slices
25        | current > end = slices
26        | otherwise = slice (current + step) end step ((current, min (current + step - 1) end) : slices)
27
28      xs = slice (World.minX world) (World.maxX world) sliceWidth []
29      ys = slice (World.minY world) (World.maxY world) sliceHeight []
30
31      xys :: [(World.Vec2, World.Vec2)]
32      xys = concatMap (\x -> map (x,) ys) xs
33
34  -- | Given slice width adn slice height, return a set of cells that are
35  -- on the borders of partition
36  partitionBorders :: Int -> Int -> World.World -> Set.HashSet (Int, Int)
37  partitionBorders sliceWidth sliceHeight world =
38    Set.union
39      (vertical (World.minX world + (sliceWidth - 1)) Set.empty)
40      (horizontal (World.minY world + (sliceHeight - 1)) Set.empty)
41    where
42      vertical :: Int -> Set.HashSet (Int, Int) -> Set.HashSet (Int, Int)
43      vertical x items
44        | x >= World.maxX world = items
45        | otherwise = vertical (x + sliceWidth) right
46        where
47          left = foldr (\y current -> Set.insert (x, y) current) items ys
```

```
48          right =
49            if x + 1 > World.maxX world
50              then left
51              else foldr (\y current -> Set.insert (x + 1, y) current) left ys
52
53      horizontal :: Int -> Set.HashSet (Int, Int) -> Set.HashSet (Int, Int)
54      horizontal y items
55        | y >= World.maxY world = items
56        | otherwise = horizontal (y + sliceHeight) above
57        where
58          below = foldr (\x current -> Set.insert (x, y) current) items xs
59          above =
60            if y + 1 > World.maxY world
61              then below
62              else foldr (\x current -> Set.insert (x, y + 1) current) below xs
63
64      ys = [World.minY world .. World.maxY world]
65      xs = [World.minX world .. World.maxX world]
66
67  fromWorld :: World.World -> Slice
68  fromWorld world =
69    Slice
70      { minX = World.minX world,
71        maxX = World.maxX world,
72        minY = World.minY world,
73        maxY = World.maxY world
74      }
```

Listing 4: Conway.PPM

```
1   module Conway.PPM (save) where
2
3   import Control.Monad (forM_)
4   import qualified Conway.World as World
5   import System.IO
6
7   cellToPixel :: Bool -> String
8   cellToPixel cell = if cell then "1" else "0"
9
10  save :: World.World -> FilePath -> IO ()
11  save world filename = do
12    withFile
13      filename
14      WriteMode
15      ( \handle -> do
16          hPutStrLn handle "P1"
17          hPutStrLn handle (show (World.width world) ++ " " ++ show (World.height world))
18
19          forM_
20            (reverse [World.minY world .. World.maxY world])
21            ( \y -> do
22                forM_
23                  [World.minX world .. World.maxX world]
24                  ( \x -> do
25                      hPutStr handle (cellToPixel $ World.getCell world (x, y))
26                  )
27
28                hPutStrLn handle ""
29            )
30      )
```

Listing 5: Conway.Simulate

```
1   {-# LANGUAGE TupleSections #-}
2
3   module Conway.Simulate (simulate, grow, simulateSync, simulateAsync) where
4
5   import Control.Parallel.Strategies
```

```
6  import qualified Conway.Partition as Partition
7  import qualified Conway.Slice as Slice
8  import Conway.World
9  import Data.Foldable
10
11 -- import Debug.Trace
12
13 type SimulateResult = (Vec2, Bool)
14
15 simulateCell :: Slice.Slice -> World -> Vec2 -> SimulateResult
16 simulateCell slice world pos@(x, y) =
17   let live = liveNeighbors slice world x y
18    in if getCell world pos
19         then
20           ( if live == 2 || live == 3
21               then (pos, True)
22               else (pos, False)
23           )
24         else
25           ( if live == 3
26               then (pos, True)
27               else (pos, False)
28           )
29
30 simulateCells :: Slice.Slice -> World -> [Vec2] -> [SimulateResult]
31 simulateCells slice oldWorld = map (simulateCell slice oldWorld)
32
33 grow :: World -> World
34 grow world = do
35   -- four corners of the expanded grid does not need to be simulated,
36   -- as they will never have 3 live neighbors in order to be alive
37
38   let xs = [(minX world) .. (maxX world)]
39       ys = [(minY world) .. (maxY world)]
40       top = map (,maxY world + 1) xs
41       bottom = map (,minY world - 1) xs
42       left = map (minX world - 1,) ys
43       right = map (maxX world + 1,) ys
44       cellsTB = top ++ bottom
45       cellsLR = left ++ right
46       growSlice = Slice.Slice {Slice.minX = minX world - 1, Slice.maxX = maxX world + 1, Slice.minY = minY world
47       growCellsTB = simulateCells growSlice world cellsTB
48       growCellsLR = simulateCells growSlice world cellsLR
49       growSize :: (Vec2, Bool) -> Int -> Int
50       growSize (_, live) count = if live then count + 1 else count
51       growSizeTB = foldr growSize 0 growCellsTB
52       growSizeLR = foldr growSize 0 growCellsLR
53       grownWorld =
54         fromWH
55           (if growSizeLR > 0 then width world + 2 else width world)
56           (if growSizeTB > 0 then height world + 2 else height world)
57    in setCells
58         (setCells grownWorld (if growSizeLR > 0 then growCellsLR else []))
59         (if growSizeTB > 0 then growCellsTB else [])
60
61 simulate :: Slice.Slice -> World -> [SimulateResult]
62 simulate slice world = do
63   let xs = [(Slice.minX slice) .. (Slice.maxX slice)]
64       ys = [(Slice.minY slice) .. (Slice.maxY slice)]
65       xys = concatMap (\x -> map (x,) ys) xs
66    in simulateCells slice world xys
67
68 simulateSync :: World -> World
69 simulateSync old =
70   runEval $ do
71     let new = setCells old (simulate (Partition.fromWorld old) old)
72         grown = grow old
73
```

```
    return $ stack new grown

simulateAsync :: Int -> Int -> Int -> World -> World
simulateAsync sliceWidth sliceHeight chunkSize old = runEval $ do
  let slices = Partition.partition sliceWidth sliceHeight old
      partitionBorders = toList $ Partition.partitionBorders sliceWidth sliceHeight old

  (slices', partitionBorders') <- parTuple2 rdeepseq rdeepseq (slices, partitionBorders)

  let grown = grow old
      sliceCells = map (`simulate` old) slices'
      partitionBorderCells =
        simulateCells
          (Partition.fromWorld old)
          old
          partitionBorders'

  (grown', partitionBorderCells', sliceCells') <-
    parTuple3
      rdeepseq
      partitionBorderCellsStrategy
      sliceCellsStrategy
      (grown, partitionBorderCells, sliceCells)

  let sliceCellWorld = setCells old (concat sliceCells')

  return $ stack (setCells sliceCellWorld partitionBorderCells') grown'
  where
    sliceCellsStrategy :: Strategy [[SimulateResult]]
    sliceCellsStrategy = parList rdeepseq

    partitionBorderCellsStrategy :: Strategy [SimulateResult]
    partitionBorderCellsStrategy = parListChunk chunkSize rdeepseq
```

Listing 6: Conway.Slice

```
module Conway.Slice where

import Control.DeepSeq

data Slice = Slice {minX :: Int, maxX :: Int, minY :: Int, maxY :: Int}
  deriving (Show, Eq)

instance NFData Slice where
  rnf slice =
    rnf (minX slice)
      `seq` rnf (maxX slice)
      `seq` rnf (minY slice)
      `seq` rnf (maxY slice)

contains :: Slice -> (Int, Int) -> Bool
contains slice (x, y) = x >= minX slice && x <= maxX slice && y >= minY slice && y <= maxY slice
```

Listing 7: Conway.World

```
module Conway.World
  ( Grid,
    Vec2,
    World (World, width, height, grid),
    guard,
    fromList,
    fromWH,
    minX,
    maxX,
    minY,
    maxY,
    liveCount,
    neighbors,
```

```
    liveNeighbors,
    getCell,
    setCell,
    setCells,
    stack,
  )
where

import Control.DeepSeq
import qualified Control.Monad as Monad
import qualified Conway.Slice as Slice
import Data.Array ((!), (//))
import qualified Data.Array as Array

type Vec2 = (Int, Int)

type Grid = Array.Array Vec2 Bool

data World = World {width :: Int, height :: Int, grid :: Grid}
  deriving (Eq)

instance Show World where
  show world =
    "width = " ++ show (width world) ++ ", height = " ++ show (height world) ++ "\n"
      ++ concatMap
        ( \y ->
            concatMap
              ( \x -> if grid world ! (x, y) then "X " else ". "
              )
              [minX world .. maxX world]
              ++ "\n"
        )
        (reverse [minY world .. maxY world])

instance NFData World where
  rnf world = rnf (width world) `seq` rnf (height world) `seq` rnf (grid world)

-- | Make sure a world has odd height and odd width
guard :: World -> IO ()
guard world = do
  Monad.guard $ odd $ height world
  Monad.guard $ odd $ width world

fromWH :: Int -> Int -> World
fromWH w h =
  World
    { width = w,
      height = h,
      grid = Array.array ((minXFromW w, minYFromH h), (maxXFromW w, maxYFromH h)) xys
    }
  where
    xs = [(minXFromW w) .. (maxXFromW w)]
    ys = [(minYFromH h) .. (maxYFromH h)]
    xys = concatMap (\x -> map (\y -> ((x, y), False)) ys) xs

-- | Convert a 2D bool list into a World
fromList :: [[Bool]] -> World
fromList rows = forEachRow rows emptyWorld (maxYFromH h)
  where
    forEachRow :: [[Bool]] -> World -> Int -> World
    forEachRow [] world _ = world
    forEachRow (r : rs) world y =
      let newWorld = forEachCol r world (minX emptyWorld) y
       in forEachRow rs newWorld (y - 1)

    forEachCol :: [Bool] -> World -> Int -> Int -> World
    forEachCol [] world _ _ = world
    forEachCol (c : cs) world x y = forEachCol cs (setCell world (x, y) c) (x + 1) y
```

```
82
83      h = length rows
84      w = length $ head rows
85      emptyWorld = fromWH w h
86
87  maxXFromW :: Int -> Int
88  maxXFromW w = w `div` 2
89
90  minXFromW :: Int -> Int
91  minXFromW w = negate $ maxXFromW w
92
93  maxYFromH :: Int -> Int
94  maxYFromH h = h `div` 2
95
96  minYFromH :: Int -> Int
97  minYFromH h = negate $ maxYFromH h
98
99  minX :: World -> Int
100 minX world = minXFromW (width world)
101
102 maxX :: World -> Int
103 maxX world = maxXFromW (width world)
104
105 minY :: World -> Int
106 minY world = minYFromH (height world)
107
108 maxY :: World -> Int
109 maxY world = maxYFromH (height world)
110
111 liveCount :: World -> Int
112 liveCount world = foldr (\cell count -> if cell then count + 1 else count) (0 :: Int) (grid world)
113
114 neighbors :: Slice.Slice -> Int -> Int -> [Vec2]
115 neighbors slice x y = filter (Slice.contains slice) raw
116   where
117     raw =
118       [ (x + 1, y),
119         (x - 1, y),
120         (x, y + 1),
121         (x, y - 1),
122         (x + 1, y + 1),
123         (x + 1, y - 1),
124         (x - 1, y + 1),
125         (x - 1, y - 1)
126       ]
127
128 liveNeighbors :: Slice.Slice -> World -> Int -> Int -> Int
129 liveNeighbors slice world x y =
130   foldr
131     ( \neighbor count ->
132         if getCell world neighbor
133           then count + 1
134           else count
135     )
136     0
137     (neighbors slice x y)
138
139 getCell :: World -> Vec2 -> Bool
140 getCell world pos@(x, y) =
141   (x >= minX world && x <= maxX world && y >= minY world && y <= maxY world)
142     && (grid world ! pos)
143
144 setCell :: World -> Vec2 -> Bool -> World
145 setCell world pos cell =
146   World
147     { width = width world,
148       height = height world,
149       grid = grid world // [(pos, cell)]
```

```
150     }
151
152 setCells :: World -> [(Vec2, Bool)] -> World
153 setCells world cells =
154   World
155     { width = width world,
156       height = height world,
157       grid = grid world // cells
158     }
159
160 -- | merge two world by layering a on top of b
161 stack :: World -> World -> World
162 stack a b = withA
163   where
164     biggerWidth = max (width a) (width b)
165     biggerHeight = max (height a) (height b)
166
167     emptyWorld = fromWH biggerWidth biggerHeight
168
169     withB = setCells emptyWorld (map (\i -> (i, getCell b i)) (Array.indices (grid b)))
170     withA = setCells withB (map (\i -> (i, getCell a i)) (Array.indices (grid a)))
```

3.3 Testing

Listing 8: test/World.hs

```
1  module World (World.test) where
2
3  import qualified Conway.World as World
4  import Test.Framework (Test, testGroup)
5  import Test.Framework.Providers.HUnit (testCase)
6  import Test.HUnit
7
8  test :: Test.Framework.Test
9  test =
10   testGroup
11     "world"
12     [ testCase
13         "world/fromList"
14         ( do
15             let world = World.fromList [[True, True, True], [True, False, True], [True, True, True]]
16
17             assertEqual "width" 3 (World.width world)
18             assertEqual "width" 3 (World.height world)
19
20             assertEqual "live count" 8 (World.liveCount world)
21         ),
22       testCase
23         "world/minX,maxX,minY,maxY"
24         ( do
25             let world = World.fromWH 5 5
26             assertEqual "" (-2) (World.minX world)
27             assertEqual "" 2 (World.maxX world)
28
29             assertEqual "" (-2) (World.minY world)
30             assertEqual "" 2 (World.maxY world)
31         ),
32       testCase
33         "world/setCell,getCell"
34         ( do
35             let world = World.fromWH 1 1
36
37             assertEqual "" False (World.getCell world (0, 0))
38
39             let newWorld = World.setCell world (0, 0) True
40             assertEqual "" True (World.getCell newWorld (0, 0))
41         ),
```

```
      testCase
        "world/stack"
        ( do
            let a = World.fromList [[False]]
            let b = World.fromList [[True, True, True]]

            let result = a `World.stack` b

            assertEqual "width" 3 (World.width result)
            assertEqual "height" 1 (World.height result)

            assertEqual "(-1, 0)" True (World.getCell result (-1, 0))
            assertEqual "(0, 0)" False (World.getCell result (0, 0))
            assertEqual "(1, 0)" True (World.getCell result (1, 0))
        )
    ]
```

Listing 9: test/Partition/Partition.hs

```
module Partition.Partition (Partition.Partition.test) where

import qualified Conway.Partition as Partition
import qualified Conway.World as World
import Test.Framework (Test, testGroup)
import Test.Framework.Providers.HUnit (testCase)
import Test.HUnit

assertValidSlices :: [Partition.Slice] -> World.World -> Assertion
assertValidSlices slices world =
  mapM_
    ( \slice -> do
        assertEqual "" True (Partition.minX slice >= World.minX world)
        assertEqual "" True (Partition.minY slice >= World.minY world)
        assertEqual "" True (Partition.maxX slice <= World.maxX world)
        assertEqual "" True (Partition.maxY slice <= World.maxY world)
    )
    slices

test :: Test.Framework.Test
test =
  testGroup
    "partition"
    [ testCase
        "partition/dividable-0"
        ( do
            let world = World.fromWH 3 3
                slices = Partition.partition 1 1 world

            assertEqual "" 9 (length slices)
            assertValidSlices slices world

            assertBool "" (Partition.Slice (-1) (-1) (-1) (-1) `elem` slices)
        ),
      testCase
        "partition/dividable-1"
        ( do
            let world = World.fromWH 3 3
                slices = Partition.partition 3 1 world

            assertEqual "" 3 (length slices)
            assertValidSlices slices world
        ),
      testCase
        "partition/dividable-2"
        ( do
            let world = World.fromWH 9 9
                slices = Partition.partition 3 3 world

```

```
            assertEqual "" 9 (length slices)
            assertValidSlices slices world

            assertBool "" (Partition.Slice (-4) (-2) (-4) (-2) `elem` slices)
        ),
      testCase
        "partition/not-dividable-0"
        ( do
            let world = World.fromWH 3 3
                slices = Partition.partition 2 1 world

            assertEqual "" 6 (length slices)
            assertValidSlices slices world
        ),
      testCase
        "partition/not-dividable-1"
        ( do
            let world = World.fromWH 3 3
                slices = Partition.partition 2 2 world

            assertEqual "" 4 (length slices)
            assertValidSlices slices world
        ),
      testCase
        "partition/not-dividable-2"
        ( do
            let world = World.fromWH 7 3
                slices = Partition.partition 2 1 world

            assertEqual "" 12 (length slices)
            assertValidSlices slices world
        )
    ]
```

Listing 10: test/Partition/PartitionBorder.hs

```
module Partition.PartitionBorder (Partition.PartitionBorder.test) where

import qualified Conway.Partition as Partition
import qualified Conway.World as World
import qualified Data.HashSet as Set
import Test.Framework (Test, testGroup)
import Test.Framework.Providers.HUnit (testCase)
import Test.HUnit

test :: Test.Framework.Test
test =
  testGroup
    "partition/border"
    [ testCase
        "partition/border/dividable-0"
        ( do
            let world = World.fromWH 3 3
                borders = Partition.partitionBorders 1 1 world
                expected =
                  [ (-1, -1),
                    (0, -1),
                    (1, -1),
                    (-1, 0),
                    (0, 0),
                    (1, 0),
                    (-1, 1),
                    (0, 1),
                    (1, 1)
                  ]

            assertEqual "size" 9 (length borders)
            assertEqual "equality" (Set.fromList expected) borders
```

```
33          ),
34        testCase
35          "partition/border/dividable-1"
36          ( do
37              let world = World.fromWH 9 9
38                  borders = Partition.partitionBorders 3 3 world
39
40              assertEqual "size" 56 (length borders)
41          ),
42        testCase
43          "partition/border/dividable-2"
44          ( do
45              let world = World.fromWH 3 3
46                  borders = Partition.partitionBorders 3 3 world
47
48              assertEqual "size" 0 (length borders)
49          ),
50        testCase
51          "partition/border/not-dividable-0"
52          ( do
53              let world = World.fromWH 5 5
54                  borders = Partition.partitionBorders 2 2 world
55                  expected =
56                    [ (-2, 2),
57                      (-1, 2),
58                      (0, 2),
59                      (1, 2),
60                      (2, 2),
61                      (-2, 1),
62                      (-1, 1),
63                      (0, 1),
64                      (1, 1),
65                      (2, 1),
66                      (-2, 0),
67                      (-1, 0),
68                      (0, 0),
69                      (1, 0),
70                      (2, 0),
71                      (-2, -1),
72                      (-1, -1),
73                      (0, -1),
74                      (1, -1),
75                      (2, -1),
76                      (-1, -2),
77                      (0, -2),
78                      (1, -2),
79                      (2, -2)
80                    ]
81
82              assertEqual "size" 24 (length borders)
83              assertEqual "equality" (Set.fromList expected) borders
84          )
85      ]
```

Listing 11: test/Simulate/Finite.hs

```
1  module Simulate.Finite (Simulate.Finite.test) where
2
3  import qualified Conway.Partition as Partition
4  import qualified Conway.Simulate as Simulate
5  import qualified Conway.World as World
6  import Test.Framework (Test, testGroup)
7  import Test.Framework.Providers.HUnit (testCase)
8  import Test.HUnit
9
10 test :: Test.Framework.Test
11 test =
12   testGroup
```

```
13      "simulate/finite"
14      [ testCase
15          "simulate/finite/0"
16          ( do
17              let world = World.fromList [[True, False, False], [False, True, True], [False, False, False]]
18
19              let newWorld = World.setCells world (Simulate.simulate (Partition.fromWorld world) world)
20                  newGrid = World.grid newWorld
21
22              assertEqual "exactly 9 cells" 9 (length newGrid)
23              assertEqual "two live cells" 2 (World.liveCount newWorld)
24              assertEqual "two live cells" True (World.getCell newWorld (0, 0))
25              assertEqual "two live cells" True (World.getCell newWorld (0, 1))
26          ),
27        testCase
28          "simulate/finite/1"
29          ( do
30              let world = World.fromList [[True, True, True], [True, True, True], [True, True, True]]
31
32              let newWorld = World.setCells world (Simulate.simulate (Partition.fromWorld world) world)
33
34              assertEqual "" 9 (length $ World.grid world)
35              assertEqual "center cell dies" False (World.getCell newWorld (0, 0))
36          )
37      ]
```

Listing 12: test/Simulate/Grow.hs

```
1   module Simulate.Grow (Simulate.Grow.test) where
2
3   import qualified Conway.Simulate as Conway
4   import qualified Conway.World as World
5   import Test.Framework (Test, testGroup)
6   import Test.Framework.Providers.HUnit (testCase)
7   import Test.HUnit
8
9   test :: Test.Framework.Test
10  test =
11    testGroup
12      "simulate/grow"
13      [ testCase
14          "simulate/grow/not-grow"
15          ( do
16              let world = World.fromList [[False, False, False], [False, False, False], [False, False, False]]
17
18              let newWorld = Conway.grow world
19
20              assertEqual "size does not grow" 3 (World.width newWorld)
21              assertEqual "size does not grow" 3 (World.height newWorld)
22              assertEqual "exactly 9 cells" 9 (length $ World.grid newWorld)
23          ),
24        testCase
25          "simulate/grow/grow-height"
26          ( do
27              let world = World.fromList [[True, True, True], [False, False, False], [False, False, False]]
28
29              let newWorld = Conway.grow world
30
31              let grid = World.grid newWorld
32
33              assertEqual "" True (World.getCell newWorld (0, 2))
34              assertEqual "" False (World.getCell newWorld (-1, 2))
35              assertEqual "" False (World.getCell newWorld (1, 2))
36
37              assertEqual "width does not grow" 3 (World.width newWorld)
38              assertEqual "height does grow" 5 (World.height newWorld)
39              assertEqual "eactly 15 cells" 15 (length grid)
40          ),
```

```
41        testCase
42          "simulate/grow/grow-width"
43          ( do
44              let world = World.fromList [[True, False, False], [True, False, False], [True, False, False]]
45
46              let newWorld = Conway.grow world
47
48              let grid = World.grid newWorld
49
50              assertEqual "" True (World.getCell newWorld (-2, 0))
51
52              assertEqual "width does not grow" 5 (World.width newWorld)
53              assertEqual "height does grow" 3 (World.height newWorld)
54              assertEqual "" 15 (length grid)
55          )
56      ]
```

Listing 13: test/Simulate/Infinite.hs

```
1  module Simulate.Infinite (Simulate.Infinite.test) where
2
3  import qualified Conway.Simulate as Conway
4  import qualified Conway.World as World
5  import Test.Framework (Test, testGroup)
6  import Test.Framework.Providers.HUnit (testCase)
7  import Test.HUnit
8
9  test :: Test.Framework.Test
10 test =
11   testGroup
12     "simulate/infinite"
13     [ testCase
14         "simulate/infinite/3x3-not-growing"
15         ( do
16             let world =
17                   World.fromList
18                     [ [True, True, False],
19                       [False, True, False],
20                       [False, True, False]
21                     ]
22
23             let syncWorld = Conway.simulateSync world
24             let asyncWorld = Conway.simulateAsync 1 1 1 world
25
26             assertEqual "width" 3 (World.width syncWorld)
27             assertEqual "height" 3 (World.height syncWorld)
28             assertEqual "# of live cells" 4 (World.liveCount syncWorld)
29             assertEqual "# of cells" 9 (length $ World.grid syncWorld)
30
31             assertEqual "" syncWorld asyncWorld
32         ),
33       testCase
34         "simulate/infinite/3x3-growing"
35         ( do
36             let world =
37                   World.fromList
38                     [ [True, True, True],
39                       [False, True, False],
40                       [False, True, False]
41                     ]
42
43             let syncWorld = Conway.simulateSync world
44             let asyncWorld = Conway.simulateAsync 1 1 1 world
45
46             assertEqual "width" 3 (World.width syncWorld)
47             assertEqual "height" 5 (World.height syncWorld)
48             assertEqual "# of live cells" 4 (World.liveCount syncWorld)
49             assertEqual "# of cells" 15 (length $ World.grid syncWorld)
```

```
50
51              assertEqual "" syncWorld asyncWorld
52          ),
53        testCase
54          "simulate/infinite/5x5"
55          ( do
56              let world =
57                    World.fromList
58                      [ [True, True, False, True, True],
59                        [False, True, False, False, False],
60                        [False, True, False, True, False],
61                        [True, True, False, True, True],
62                        [False, True, False, False, False]
63                      ]
64
65              -- print world
66
67              let syncWorld = Conway.simulateSync world
68              let asyncWorld = Conway.simulateAsync 3 3 1 world
69
70              -- print syncWorld
71              -- print (World.grid asyncWorld)
72
73              assertEqual "# of live cells" 16 (World.liveCount syncWorld)
74              assertEqual "width" 5 (World.width syncWorld)
75              assertEqual "width" 5 (World.height syncWorld)
76
77              assertEqual "width" (World.width syncWorld) (World.width asyncWorld)
78              assertEqual "height" (World.height syncWorld) (World.height asyncWorld)
79
80              assertEqual "same number of living cells" (World.liveCount syncWorld) (World.liveCount asyncWorld)
81              assertEqual "same number of cells" (length $ World.grid syncWorld) (length $ World.grid asyncWorld)
82              assertEqual "equality" syncWorld asyncWorld
83          ),
84        testCase
85          "simulate/infinite/9x9"
86          ( do
87              let world =
88                    World.fromList
89                      [ [True, True, False, True, False, True, False, True, True],
90                        [True, True, False, True, False, True, False, True, True],
91                        [True, True, False, True, False, True, False, True, True],
92                        [True, True, False, True, False, True, False, True, True],
93                        [True, True, False, True, False, True, False, True, True],
94                        [True, True, False, True, False, True, False, True, True],
95                        [True, True, False, True, False, True, False, True, True],
96                        [True, True, False, True, False, True, False, True, True],
97                        [True, True, False, True, False, True, False, True, True]
98                      ]
99
100             let syncWorld = Conway.simulateSync world
101             let asyncWorld = Conway.simulateAsync 3 3 1 world
102
103             assertEqual "" (length $ World.grid syncWorld) (length $ World.grid asyncWorld)
104             assertEqual "" syncWorld asyncWorld
105         )
106     ]
```