Parallel Functional Programming Final Project

Tian Low (ttl2132) Selena Huang (sh3696)

December 2020

1 Introduction

The project we worked on is a nonogram solver in Haskell. A nonogram, also known as picross, is a rectangle filled with squares that a user will either shade or not shade in, typically in order to complete a picture. The user is given a set of numbers in each row that state how many squares are filled in and in what order. If there is more than one number in the set, this indicates that there is at least one blank square in between both number of squares. Similar to the Sudoku solver we saw in class, we wanted to see if we could parallelize the algorithm on a set of puzzles. We chose nonograms because both of us enjoy these games, and it is a relaxing and aesthetic game.

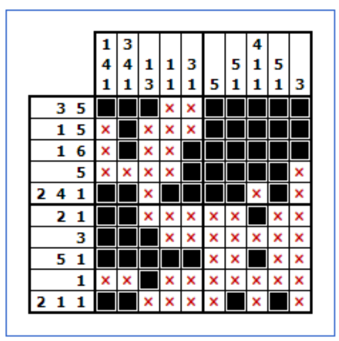


Figure 1: An example of a nonogram

1.1 Nonogram Algorithm

The nonogram algorithm we started with was a basic algorithm found on the HaskellWiki site[1]. The algorithm to solve a nonogram is similar to that of Sudoku. It begins by storing all the possible values for each cell, though in this case there are only two: filled or not filled. These sets are iteratively reduced until there is only one value left, and the cell is then assigned that value. If there are no cells that can be reduced, a guess is made and the puzzle is split into two. If a puzzle ends in a contradiction, it is discarded, and if it is successfully completed, it is collected as a solution.

1.2 Puzzle Data Collection

Because it would be hard see any differences in time in parallelizing one puzzle, we decided to parallelize the process of solving a collection of puzzles. We searched for datasets of nonograms stored in text files to parse. We found a database on github: mikix/nonogram-db[2] as well as a website that has a database of user created puzzles[3], that could be exported[4] and collected them into our own puzzle directory, which our main method would then read through and parse to create a puzzle. From there, we implemented the basic nonogram solving algorithm, and counted the number of successful puzzles. Our puzzle directory currently has 82 puzzles.

2 Implementation

To understand better how parallelism works and see if multiple cores actually improve our times, we first solved all puzzles sequentially before moving on to parallelism. This is so we have a baseline to start with.

2.1 Sequential Solution

We wanted to see how long it would take for one core to sequentially solve all the puzzles. The snippet below shows the logic flow of our main sequential method. This ensures that each puzzle is actually called and solved. The full listing of the code is in the Code Listing section.

```
function nonogram_solver(file_contents):
    get (horizontal, vertical) from file_contents
    solve_puzzle (horizontal, vertical)
    return True if solved, False otherwise

function main:
    puzzle_directory = the path to puzzle directory
    files = all *.non files in puzzle_directory
    contents = read files
    solutions = []
    for each file_content in contents:
        solutions.append( nonogram_solver(file_content))
    return length of solutions
```

For our sequential solution, we can see that it takes around 10.5s for the operation to complete for 50 puzzles. The following Threadscope screenshot shows that with multiple cores, the activity moves into the other threads and is spread more evenly.

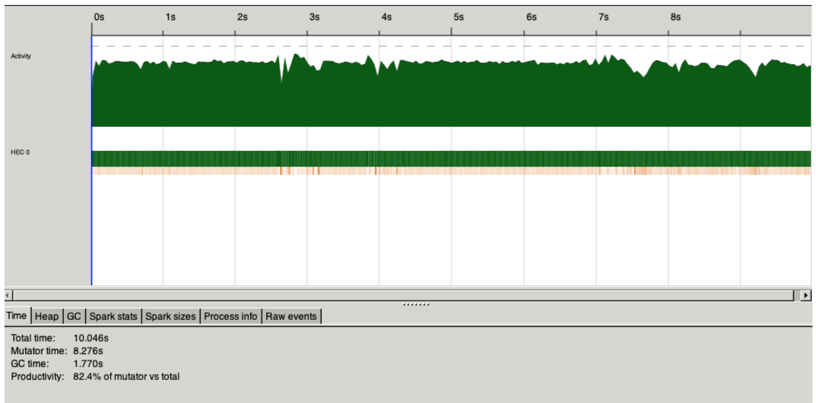


Figure 2: Threadscope of the sequential solution on 1 core, 50 puzzles

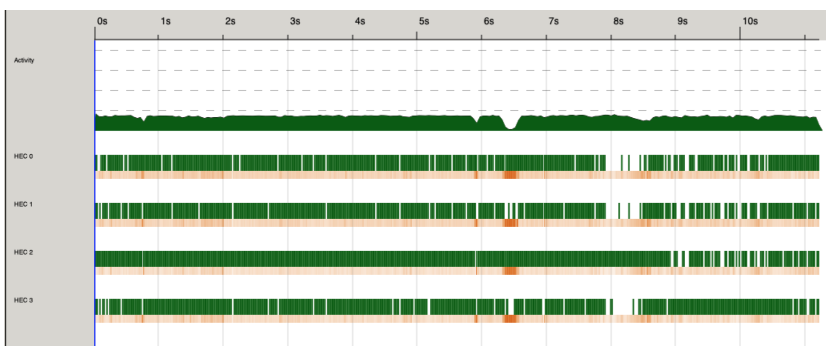


Figure 3: Threadscope of the sequential solution on 4 cores, 50 puzzles

However, as the number of cores increase, there is actually an increase in the total time taken due to an increase in time taken for garbage collection and a constant mutator time. There is also a decrease in productivity.

This confirms the fact that even if the processes are running on separate threads, overall, running a sequential process on multiple cores does not make the process more efficient.

2.2 Parallelism

We decided to try different functions from the Control.Parallel.Strategies package, namely

- withStrategy
- rpar
- rparWith
- rdeepseq
- parMap
- parBuffer

Here are some examples of the strategies we took to reach our final result.

1. First, we tried parallelizing by the horizontal and vertical grids from getGrid with rpar, but there was very sparse activity after 5 ms and had no parallelization.

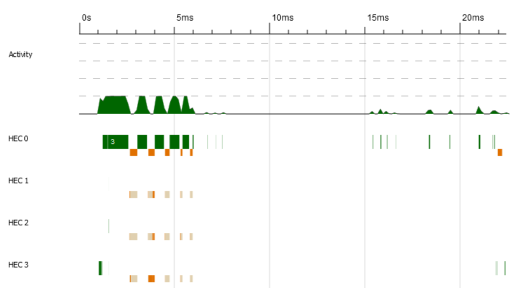


Figure 4: Split on horizontal and vertical grid

2. We then tried parallelizing using parPair (shown as a comment in the code listing) in beforeAfter, as it seemed that there was not much computation to do for getGrid; we believed that using parallelization on a larger computation could result in a more balanced and parallel algorithm. However, this seemed to take much more time and still had low activity throughout.

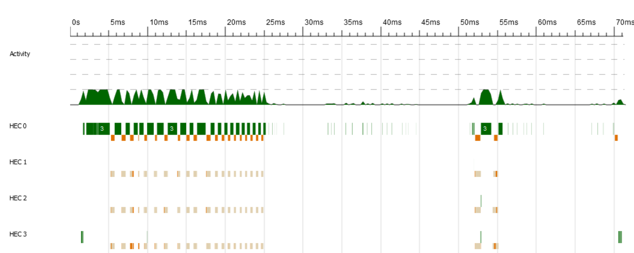


Figure 5: Utilizing parpair in beforeAfter only

3. Combining both resulted in a time somewhere in the middle between both steps. Interestingly, combining both strategies seemed to make the activity more balanced throughout, but it still had the same pattern of no parallelization.

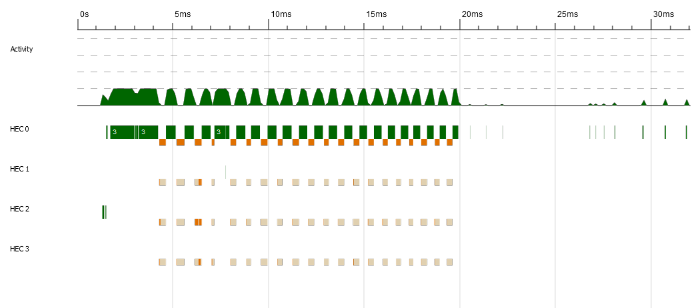


Figure 6: Combining both strategies from attempts 1 and 2

4. Consequently, we moved onto try using rdeepseq on beforeAfter. With this change, the amount of activity was still not parallelized, but the overall activity for the single thread seemed to be more balanced throughout (compared to mainly doing computation in the first of the time taken in the previous attempt). However, this strategy took much more time, possible due to additional garbage collection.

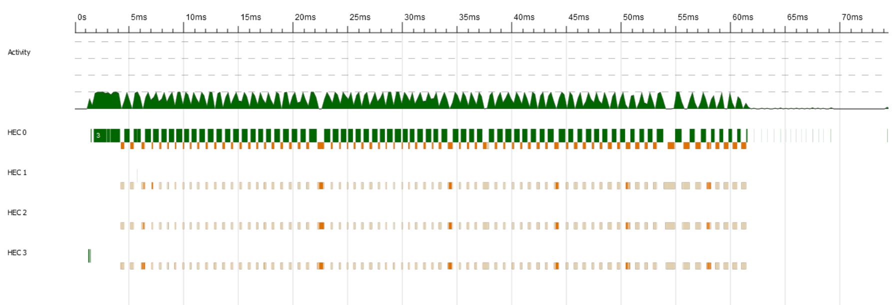


Figure 7: rdeepseq on beforeAfter

5. Trying with the same parPair strategy with rdeepseq instead of rpar on beforeAfter resulted in a very odd pattern, in which there seemed to be some overlapped parallelization at the beginning and end, but the middle, only one core was used.

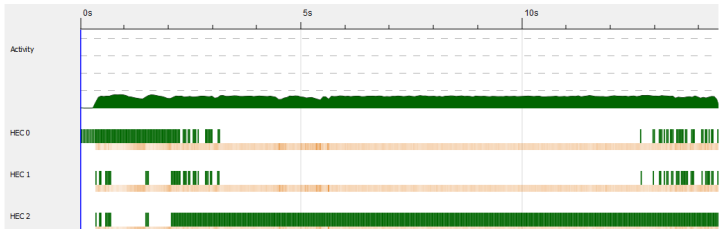


Figure 8: parPair with rdeepseq instead of rpar on beforeAfter

6. Using the strategy parPair on lineStepFwd seemed to increase the activity in the middle of the program to 16% parallelization. One can see in the graph that all four cores have activity, but unfortunately, for the most part, the total amount of activity remained low, indicating that there still was not enough parallelization.

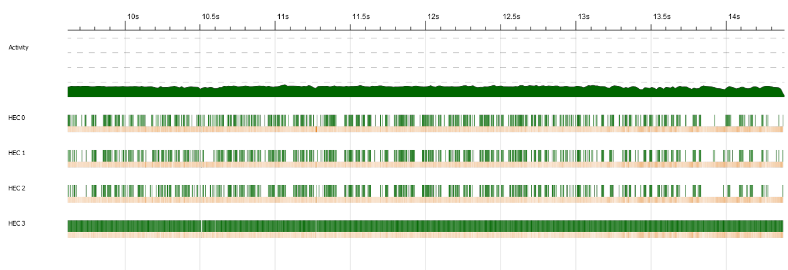


Figure 9: Using the strategy parPair for lineStepFwd

7. Because we were running the algorithm on 50 puzzles, we decided to try using parMap rpar on the 50 puzzles in Main.hs. The parallelization increased from 17% to 37%, which was a large jump. One can also see a sudden spike in activity at the start of the program in Threadscope. The time also decreased to be slightly more than half the time without parMap. Below, one can see the program being run on 2 vs. 4 cores.

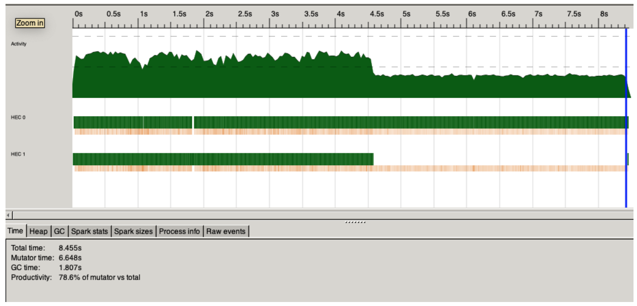


Figure 10: Using the strategy parMap rpar with 2 cores for 50 puzzles

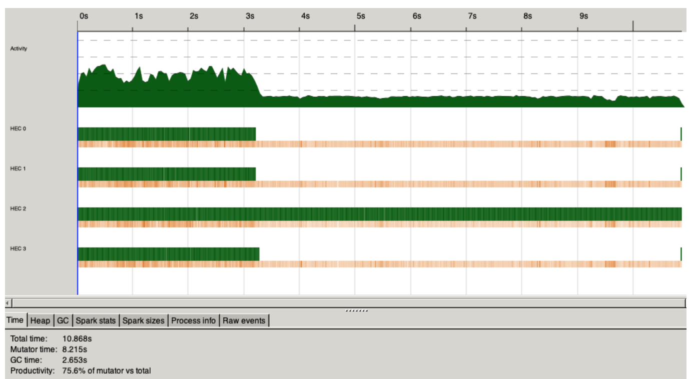
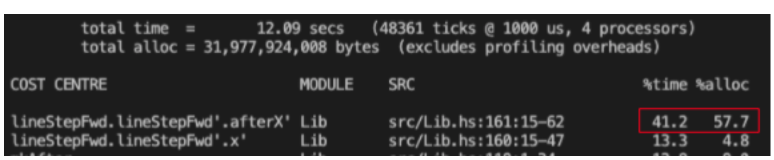


Figure 11: Using the strategy parMap rpar with 4 cores for 50 puzzles

8. The portion with a large amount of activity became a quarter of the time in four cores, which led us to believe that there was a sequential portion of the algorithm, and/or there was a particular puzzle that was taking much longer to computer than others. Consequently, we decided to increase the number of puzzles to 82. We also checked the profiling tools, noticing that 42% of the time was spent on a single step. Thus, we decided to work on parallelizing this step.



```
        total time  =       12.09 secs   (48361 ticks @ 1000 us, 4 processors)
        total alloc = 31,977,924,008 bytes  (excludes profiling overheads)

COST CENTRE                          MODULE    SRC                          %time %alloc

lineStepFwd.lineStepFwd'.afterX'    Lib       src/Lib.hs:161:15-62          41.2   57.7
lineStepFwd.lineStepFwd'.x'         Lib       src/Lib.hs:160:15-47          13.3    4.8
```

Figure 12: Screenshot of time distribution

9. The first strategy we tried was using parMap rpar in the this afterX'

variable. Between the following two figures, one can see the increase in time where overall activity was high, namely increasing the parallelization from 40% to 44%.

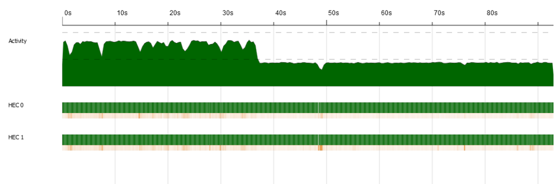


Figure 13: Without changing afterX'

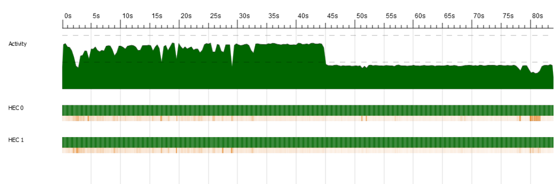


Figure 14: After changing afterX' to use parMap rpar

10. Ultimately, we used parBuffer 100 in afterX', as well as parMap rdeepseq in Main.hs. The parallelization increased to almost 50% with these changes. Using 3 cores drastically decreased the time to 18 seconds, but using 2 cores caused higher productivity for both cores. However, 4 cores was worse than both in time and performance, and had a drastic dip in activity at 20 seconds. We also rechecked the time distribution with the profiling tool, and the total percent of time decreased from 41.2 to 21.6. The allocation percentage allocation also decreased from 57.7 to 32.2.

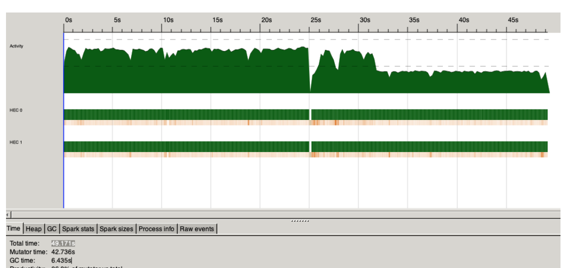


Figure 15: parBuffer 100 on 2 cores

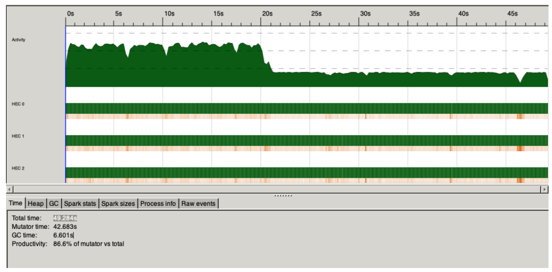


Figure 16: parBuffer 100 on 3 cores

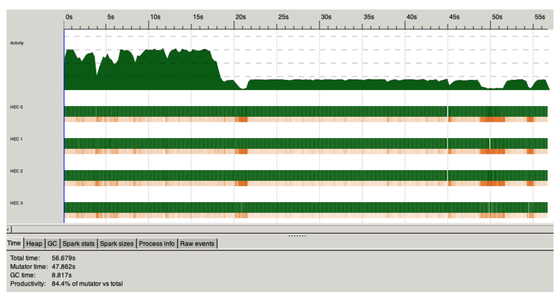
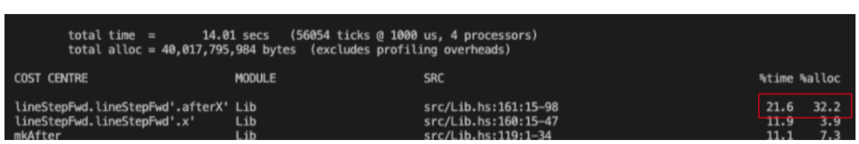


Figure 17: parBuffer 100 on 4 cores



```
        total time  =       14.01 secs   (56054 ticks @ 1000 us, 4 processors)
        total alloc = 40,017,795,984 bytes  (excludes profiling overheads)

COST CENTRE                     MODULE      SRC                         %time %alloc

lineStepFwd.lineStepFwd'.afterX' Lib        src/Lib.hs:161:15-98         21.6   32.2
lineStepFwd.lineStepFwd'.x'      Lib        src/Lib.hs:160:15-47         11.9    3.9
mkAfter                          Lib        src/Lib.hs:119:1-34          11.1    7.3
```

Figure 18: Final screenshot of time distribution for parBuffer

3 Conclusion

3.1 Settings

We ran the code on a dual-core Macbook Pro 2017 to produce these results.

3.2 Analysis

The initialization of multiple puzzles can be processed in parallel, which ultimately decreased the time from 80 seconds to 18 seconds, which is drastic jump. However, in our solution, the end of the Threadscope graph consistently dropped in overall activity in the end. We believe that some parts of the algortihm were run sequentially, namely where we iterated steps. This may have cause the algorithm to run serially after parallelizing the initial computations. There were more garbage collection and fizzled sparks than desired due to many sparks being generated, but there were 0 overflowing sparks, which led us to believe that this was an overall okay parallelization.

3.3 Problems

While working on this project, we encountered a variety of different problems. The first was that the time taken is exponentially shorter using two or four cores, but the efficiency doesn't increase. In addition, we saw that one core was not receiving tasks while the others were split evenly. The amount of garbage collection happening during program run time was a significant portion as well. One other problem was that the run times differed significantly when running them between our computers, as well as at different times. Finally, we found that despite parallelizing the step that took the most time, adding more cores didn't decrease the total time taken.

3.4 Performance

In terms of performance, we could see pretty obviously that parallelizing the main got better results when on two cores, but the GC balances out the time reduced at four cores. When we parallelized the step that took the highest percentage of time, we made it exponentially faster, with more parallelization, but the productivity decreased, and the sparks were not well balanced.

3.5 Final Words

This project was very interesting but difficult. We we were able about the various methods and strategies of parallelism, but found out that there was much more we still couldn't understand, even with Threadscope and the profiling tools. The hardest part was seeing that something was wrong but not being able to find the place that was producing the errors. If there was more time, an interesting direction to pursue would be working with larger puzzles or more puzzles to see if the problems were due to the specific puzzles or certain parts within the algorithm.

4 Code Listing

```haskell
module Main where

import Lib
import System.Directory (getDirectoryContents)
import Control.Parallel.Strategies (parMap, rdeepseq)


-- Second main: Sequentially reads all the contents of all the files
-- Reads all the puzzles in the absolute path because Haskell sucks
main :: IO()
main = do
    let path = "C:/Users/chiyo/Desktop/nonogram/puzzles/"
    files <- getDirectoryContents path
    let onlyFiles = filter (`notElem` [".",".."]) files
    let absoluteFiles = map (path ++ ) onlyFiles
    contents <- mapM readFile absoluteFiles
    let solutions = parMap rdeepseq nonogram contents  -- solutions is of type [Bool]
    print (length (filter (== True)  solutions ))
```

Listing 1: app/Main.hs

```haskell
module Lib where

import Data.List.Split(splitOn)
import qualified Data.Set as Set
import Data.Set (Set)
import qualified Data.Map as Map
import Data.Map (Map)
import Data.List
import Control.Parallel.Strategies(NFData, rpar, withStrategy, parBuffer, rdeepseq, parList, using)

-- ----------------------------------------------------------------------

-- Parsing
-- parses the Ints from the Chars
clean :: [Char] -> [Int]
clean row = map (\word -> read word::Int) $ splitOn "," row

-- reads in the content of the file, outputs True if puzzle is solved, False otherwise
nonogram :: String -> Bool
nonogram puzzle_board =
    let info = init.tail $ dropWhile (/="") $ lines puzzle_board in
```

```haskell
    let h = map (\line -> clean line ) $ tail $ takeWhile (/= "") info in
    let v = map (\line -> clean line ) $ tail $ filter (/= "") (dropWhile (/= "") info) in
    check $ solve (puzzle h v)

-- ----------------------------------------------------------------------
-- Cells

newtype Value = Value Int
    deriving (Eq, Ord, Show)

-- | Negative values encode empty cells, positive values filled cells
empty :: Value -> Bool
empty (Value n) = n <= 0

full :: Value -> Bool
full = not . empty

type Choice = Set Value

-- ----------------------------------------------------------------------
-- Puzzle

type Grid = [[Choice]]

-- | Datatype for solved and unsolved puzzles
data Puzzle = Puzzle
    -- | List of rows, containing horizontal choices for each cell
    { gridH :: Grid
    -- | List of columns, containing vertical choices for each cell
    , gridV :: Grid
    -- | What is allowed before/after a specific value?
    --   (after (Value 0)) are the values allowed on the first position
    , afterH, beforeH :: [Value -> Choice]
    , afterV, beforeV :: [Value -> Choice]
    }

instance Eq Puzzle where
    p == q = gridH p == gridH q

instance Show Puzzle where
    show = dispGrid . gridH

-- | Transpose a puzzle (swap horizontal and vertical components)
transposeP :: Puzzle -> Puzzle
transposeP p = Puzzle
    { gridH      = gridV p
    , gridV      = gridH p
    , afterH     = afterV p
    , beforeH    = beforeV p
```

```haskell
    , afterV      = afterH p
    , beforeV     = beforeH p
    }

-- | Display a puzzle
dispGrid :: [[Set Value]] -> [Char]
dispGrid = concatMap (\r -> "[" ++ map disp'' r ++ "]\n")
 where disp'' x
        | Set.null      x = 'E'
        | setAll full   x = '1'
        | setAll empty  x = '0'
        | otherwise       = '/'


-- ----------------------------------------------------------------------
-- Making puzzles

-- | Generate puzzle
puzzle :: [[Int]] -> [[Int]] -> Puzzle
puzzle h v = Puzzle
    { gridH   = gH
    , gridV   = gV
    , afterH  = fst abH
    , beforeH = snd abH
    , afterV  = fst abV
    , beforeV = snd abV
    }
 where rows = length h
       cols = length v
       ordersH = map order h
       ordersV = map order v
       (abH, abV) = (beforeAfter ordersH, beforeAfter ordersV)
       (gH, gV) = (getGrid cols ordersH, getGrid rows ordersV)

getGrid :: Ord a => Int -> [[a]] -> [[Set a]]
getGrid numCells orders = map(replicate numCells . Set.fromList) orders

beforeAfter :: [[Value]] -> ([Value -> Choice], [Value -> Choice])
beforeAfter orders = (after, before)
    where before = map mkAfter $ map reverse orders
          after = map mkAfter orders

-- | Gets possible values for a line in order
order :: [Int] -> [Value]
order = order' 1
 where order' n []   = [Value (-n), Value (-n)]
       order' n (x:xs) = [Value (-n), Value (-n)] ++ map Value [n..n+x-1] ++ order' (n+x) xs

mkAfter :: [Value] -> Value -> Choice
mkAfter ord = (mkAfterM ord Map.!)

mkAfterM :: [Value] -> Map Value (Set Value)
mkAfterM ord  = Map.fromListWith (Set.union) aftersL
```

```haskell
 where aftersL =
                 (if length ord > 2
                 then [(Value 0, Set.singleton (ord !! 2))]
                 else []) ++
               zip (Value 0:ord) (map Set.singleton ord)

-- ----------------------------------------------------------------------
-- Checking puzzles

check :: [Puzzle] -> Bool
check ps
    | length ps == 0    = False
    | invalid $ head ps = False
    | done $ head ps    = True
    | otherwise         = False

done :: Puzzle -> Bool
done = all (all ((==1) . Set.size)) . gridH

invalid :: Puzzle -> Bool
invalid = any (any Set.null) . gridH

-- ----------------------------------------------------------------------
-- Algorithm Stepping

-- | Deterministic solving
solveD :: Puzzle -> Puzzle
solveD = consecSame . iterate step

-- | Combine steps
step :: Puzzle -> Puzzle
step = hvStep . transposeP . lineStep . transposeP . lineStep

-- | Single step
lineStep :: Puzzle -> Puzzle
lineStep p = p { gridH = gridH'' }
 where gridH'  = zipWith lineStepFwd (afterH  p) (gridH p)
       gridH'' = zipWith lineStepBack (beforeH p) (gridH')

-- | lineStep on a single line forward and backward
lineStepFwd :: (Value -> Set Value) -> [Set Value] -> [Set Value]
lineStepFwd after row = lineStepFwd' (after (Value 0)) row
 where lineStepFwd' _         []     = []
       lineStepFwd' afterPrev (x:xs) = x' : lineStepFwd' afterX' xs
        where x' = Set.intersection x afterPrev
              afterX' = Set.unions $ withStrategy (parBuffer 100 rpar) $ map after $ Set.toList x'

lineStepBack :: (Value -> Set Value) -> [Set Value] -> [Set Value]
lineStepBack before = reverse . lineStepFwd before . reverse

-- | Sharing information between the horizontal grid and vertical grid
```

```haskell
hvStep :: Puzzle -> Puzzle
hvStep p = p { gridH = gridH', gridV = transpose gridV't }
 where (gridH', gridV't) = zMap (zMap singleStep) (gridH p) (transpose (gridV p))

-- Step on a single cell
singleStep :: Set Value -> Set Value -> (Set Value, Set Value)
singleStep h v = filterCell empty . filterCell full $ (h,v)

-- Step on a single cell, for a single condition, if either h or v satisfies the condition
-- then the other is filtered so it will satisfy as well
filterCell :: (a -> Bool) -> (Set a, Set a) -> (Set a, Set a)
filterCell cond (h,v)
    | setAll cond h = (h, Set.filter cond v)
    | setAll cond v = (Set.filter cond h, v)
    | otherwise     = (h, v)

-- ----------------------------------------------------------------------

-- Nondeterministic

-- | Solve a puzzle, gives all solutions
solve :: Puzzle -> [Puzzle]
solve p
 | all (all ((==1) . Set.size)) . gridH $ p' = [p'] -- single solution
 | invalid p' = []         -- no solutions
 | otherwise       = concatMap solve (guess p') -- we have to guess
 where p' = solveD p

-- | Branch out to multiple possible choices for grids

guess :: Puzzle -> [Puzzle]
guess p = map (\gh -> p {gridH = gh} ) gridHs
 where gridHs = getMultiple (getMultiple getChoices) (gridH p)

-- | Gets multiple possible choices for a single cell
getChoices :: Choice -> [Choice]
getChoices = map Set.singleton . Set.toList

-- | Tries to split a single item in a list using the function f
--   Stops at the first position where f has more than 1 result.
getMultiple :: (a -> [a]) -> [a] -> [[a]]
getMultiple _ []     = []
getMultiple f (x:xs)
 | length fx  > 1 = map (:xs) fx
 | length fxs > 1 = map (x:) fxs
 | otherwise      = []
 where fx  = f x
       fxs = getMultiple f xs

-- ----------------------------------------------------------------------

-- Utilities
```

```haskell

-- | parallelization, especially on zMap
par' :: NFData a => [a] -> [a]
par' = (`using` parList rdeepseq)

-- Examples of some other strategies that we tried
-- parPair2 = do
--   evalTuple2 (rparWith rdeepseq) (rparWith rdeepseq)

-- parRds :: NFData a => [a] -> [a]
-- parRds = (`using` parBuffer 250 rdeepseq)

-- parPair :: Strategy (a,b)
-- parPair (a,b) = do
--    a' <- rpar a
--    b' <- rpar b
--    return (a',b')

-- | Set.all, similar to Data.List.all
setAll :: (a -> Bool) -> Set a -> Bool
setAll f = all f . Set.toList

-- | A zip-like map
zMap :: (a -> b -> (c, d)) -> [a] -> [b] -> ([c], [d])
zMap f a b = unzip $ zipWith f a b

-- | Find the first item in a list that is repeated
consecSame :: Eq a => [a] -> a
consecSame (a:b:xs)
 | a == b    = a
 | otherwise = consecSame (b:xs)

consecSame _ = error "Invalid"
```

Listing 2: src/Lib.hs

```haskell
module Main where

import Test.HUnit
import Lib
import qualified Data.Set as Set

testE2E :: Test
testE2E = TestCase (do
        content <- readFile "C:/Users/chiyo/Desktop/nonogram/puzzles/1.non"
        let info = init.tail $ dropWhile (/="") $ lines content
        let h = map (\line -> clean line ) $ tail $ takeWhile (/= "") info
        let v = map (\line -> clean line ) $ tail $ filter (/= "") (dropWhile (/= "") info)
        assertEqual "horizontal grid," [[2],[2,1],[1,1],[3],[1,1],[1,1],[2],[1,1],[1,2],[2]] h
        assertEqual "vertical grid," [[2,1],[2,1,3],[7],[1,3],[2,1]] v
        assertEqual "solution for puzzle 1.non," "[01100]\n[01101]\
```

```haskell
n[00101]\n[01110]\n[10100]\n[10100]\n[00110]\n[01010]\n[01011]\n[11000]\n" $ show $ head $ solve (puzzle h v))

testInvalid :: Test
testInvalid = TestCase (do
        assertEqual "test invalid puzzle" "False" $ show $ check $ solve (puzzle [[2],[2]] [[5],[5]])
    )

testOrder :: Test
testOrder = TestCase (do
        assertEqual "possible line values" (map Value [-1,-1, 1, -2,-2, 2,3, -4,-4, 4,5,6, -7,-7, 7, 8, 9, 10, -11, -11]) $ order [1,2,3,4]
    )

testFilterCell :: Test
testFilterCell = TestCase (do
        let filterSol = Set.fromList $ map Value [2]
        let noneFiltered = Set.fromList $ map Value [2,1]
        assertEqual "filtering cells" (filterSol, noneFiltered) $ filterCell full (Set.fromList $ map Value [-8,-7,-1,2], noneFiltered)
    )

testSS :: Test
testSS = TestCase (do
        let filterSol = Set.fromList $ map Value [-2,2,3]
        let noneFiltered = Set.fromList $ map Value [-2]
        assertEqual "double filtering single cell" (noneFiltered, noneFiltered) $ singleStep filterSol noneFiltered
    )

testZMap :: Test
testZMap = TestCase (do
        let sol = ["ad", "bcef"]
        let result = zMap (\x y -> (x++y, x++y)) ["a", "bc"] ["d", "ef"]
        assertEqual "simply zip map example" (sol, sol) result
    )

testConsecSame :: Test
testConsecSame = TestCase (do
        let p1 = (puzzle [[1],[2],[3,4]] [[1],[2],[3,4]])
        let p2 = (puzzle [[5],[9],[3,4]] [[5],[9],[3,4]])
        assertEqual "only first consecutive puzzles are returned" p1 $ consecSame [p1, p1, p2, p2]
    )

tests :: Test
tests = TestList [TestLabel "testE2E" testE2E, TestLabel "testOrder" testOrder, TestLabel "testFilterCell" testFilterCell,
                  TestLabel "testSS" testSS, TestLabel "testZMap" testZMap, TestLabel "testInvalid" testInvalid ]

main :: IO Counts
main = do
```

```
        runTestTT tests
```

Listing 3: test/Spec.hs

5 References

1. https://wiki.haskell.org/Nonogram
2. https://github.com/mikix/nonogram-db
3. https://webpbn.com/
4. https://webpbn.com/export.cgi