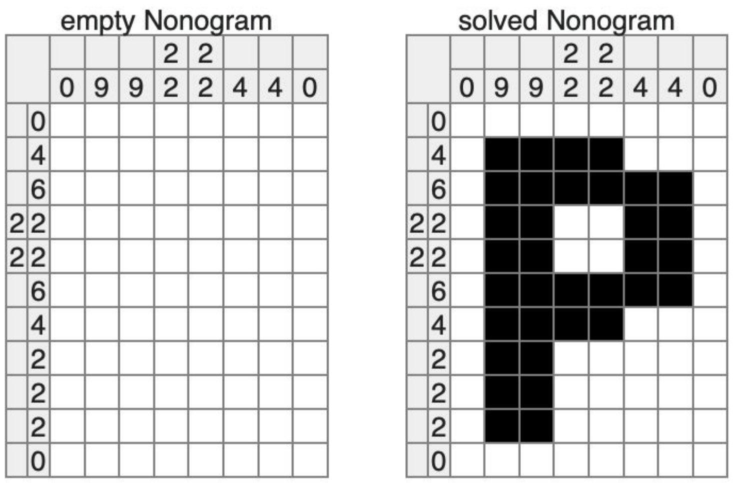
Manav Goel mg3851
Tanvi Hisaria th2720

Parallelized Nonogram Solver

Nonograms:

A nonogram is a logic puzzle similar to sudoku. You have a grid of squares, which must be either filled in or left blank. Beside each row of the grid are hints that list the lengths of the runs of black squares on that row. Above each column are listed the lengths of the runs of black squares in that column. The aim is to find all black squares, which usually reveals some sort of hidden picture at the end. The puzzle can be of various different sizes, and either a square or a rectangle. Here is an example:



Problem Description:

For this project, we decided to implement a simple search based algorithm with backtracking, and parallelized it. Then, we decided to test our program on inputs of different sizes (5X5, 10X10, and 20X20) to see whether we get a significant increase in speed by parallelizing our code.

Algorithm:

We implemented a backtracking algorithm that builds a completed puzzle top to bottom, left to right, by building the rows starting from the top and checking if they match the signatures of the columns.

On any iteration of the recursive backtracking, if we have hints remaining in a row, we can either choose to try and insert the corresponding number of black cells into the row, or insert a white cell instead. Whenever we try to insert a Color, we can check against the respective columns to see if it fits. If it does, we recurse deeper, otherwise we return. This is part we were able to parallelize, choosing to add the black cells or a white one. We execute both options in parallel by creating a new spark (using a par call) for each branch, and keep the result that ends up yielding a finished puzzle.

If there are no hints left in the row, we begin trying to build the next row. If there are no row hints left in the puzzle at all, we can check to see if the column hints are empty, meaning that the puzzle is solved.

Results:
For each of the three puzzles, we observed the following times:

5 X 5 puzzle

	Sequential	**2 cores**	**4 cores**
real	0.028s	0.027s	0.028s
user	0.000s	0.016s	0.000s
sys	0.031s	0.000s	0.031s

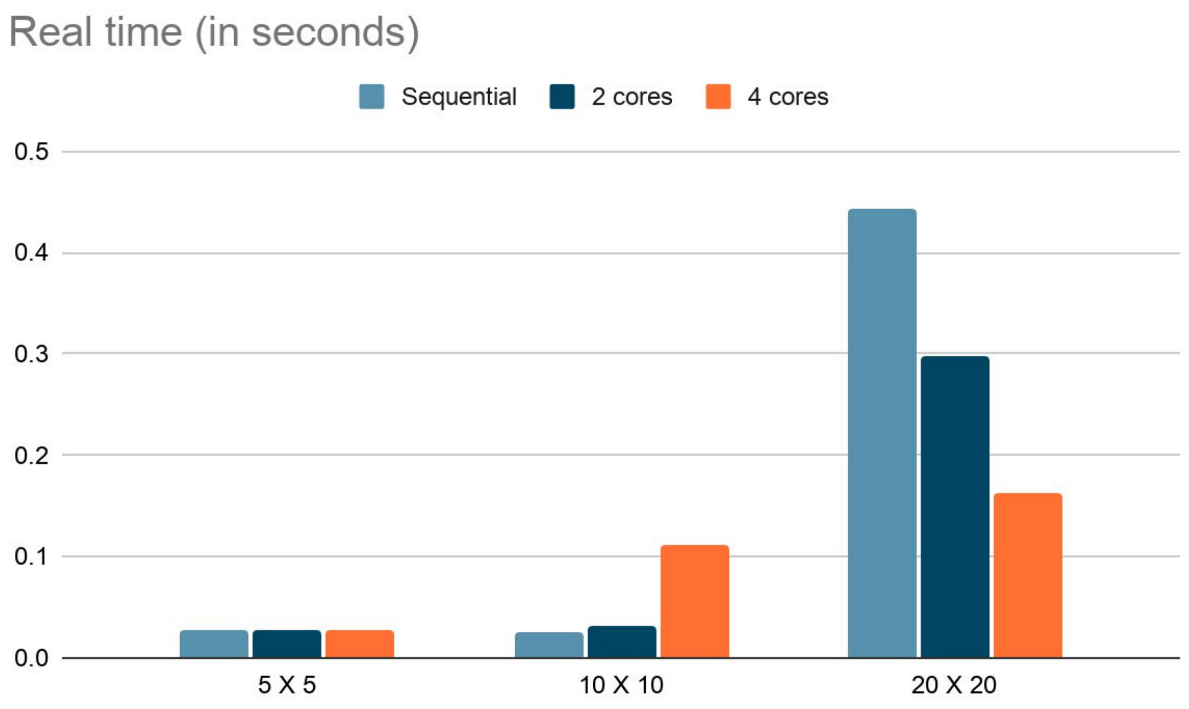
10 X 10 puzzle

	Sequential	**2 cores**	**4 cores**
real	0.026s	0.032s	0.111s
user	0.000s	0.000s	0.031s
sys	0.016s	0.031s	0.031s

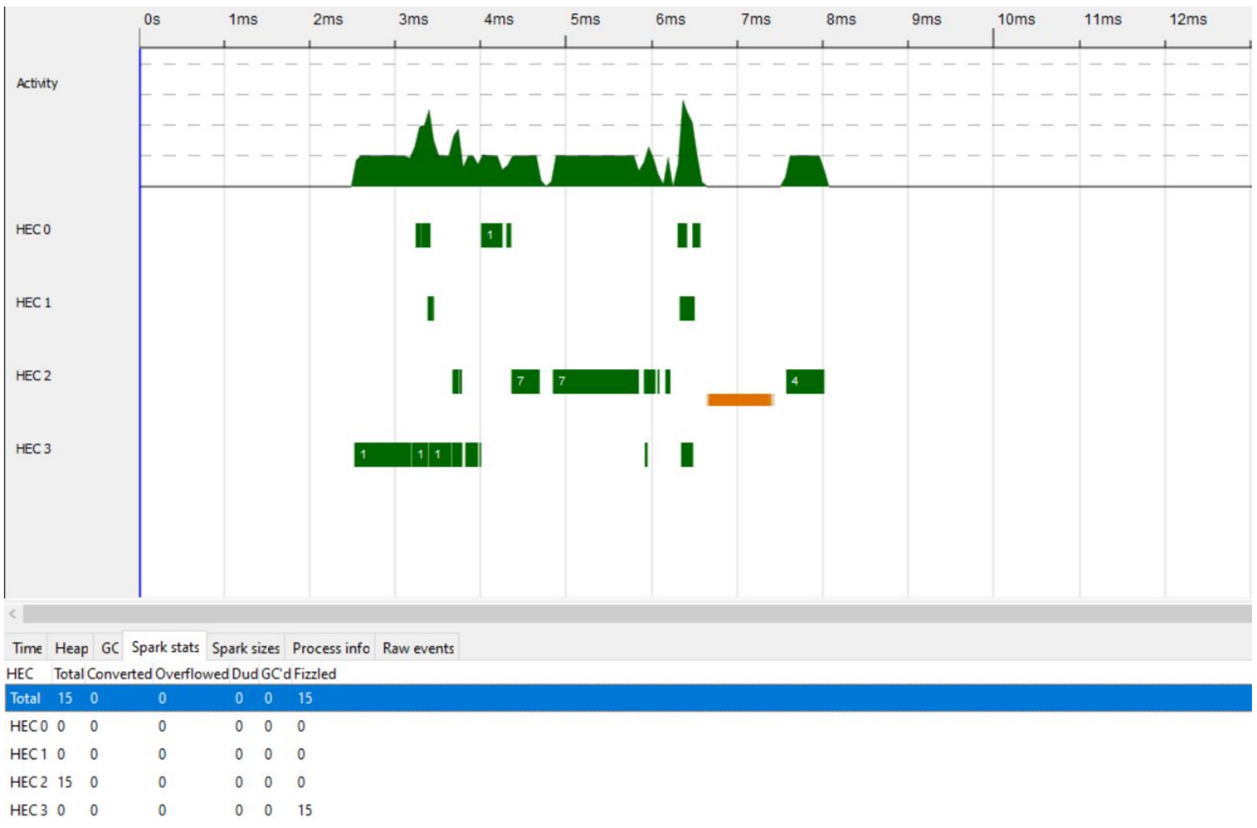
20 X 20 puzzle

	Sequential	**2 cores**	**4 cores**
real	0.444s	0.298s	0.162s
user	0.375s	0.453s	0.500s
sys	0.047s	0.078s	0.063s

While the speedup is hard to gauge from these numbers, the following chart demonstrates it better:



We see that for the two smaller puzzles, the real time either increases or stays the same. This is probably because the overhead costs introduced by parallelisation outweigh the benefits of parallelisation for the smaller puzzles. On looking at the spark statistics on threadscope, we observed the following:

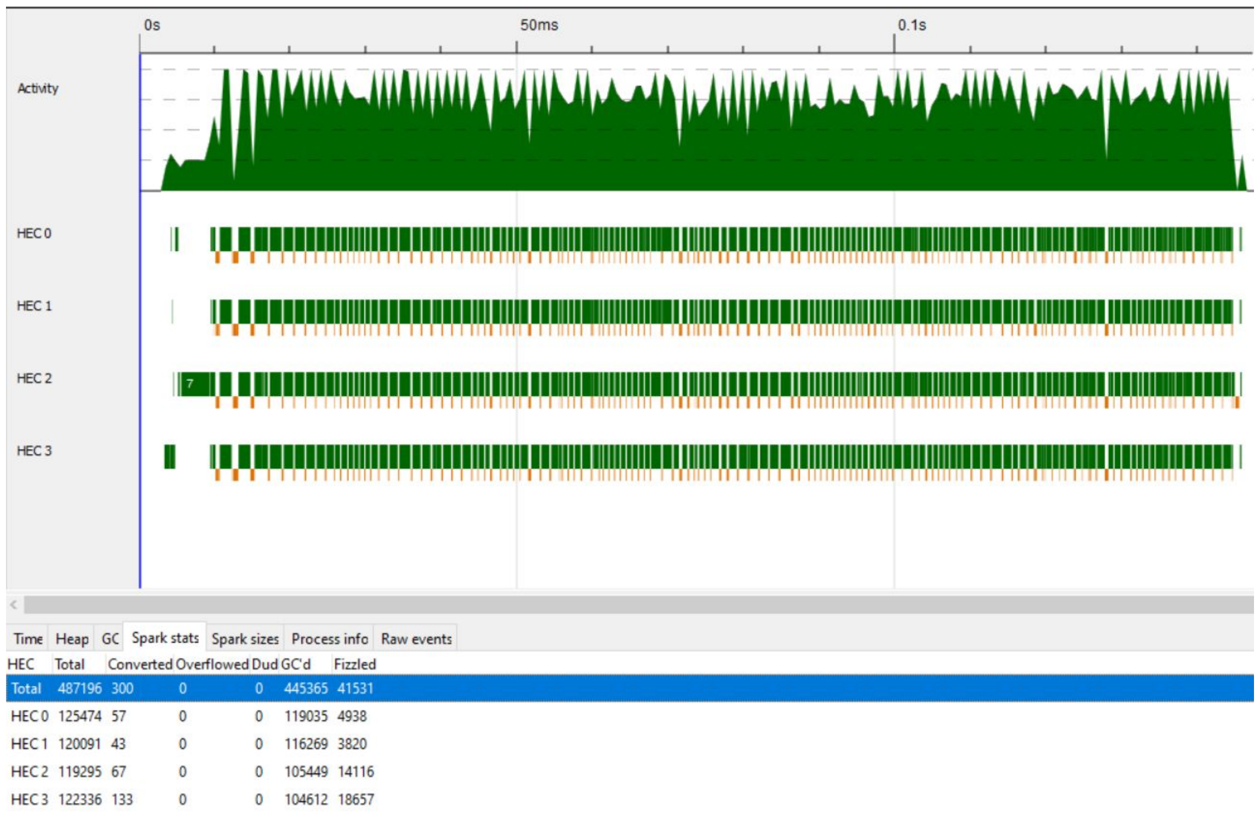


HEC	Total	Converted	Overflowed	Dud	GC'd	Fizzled
Total	15	0	0	0	0	15
HEC 0	0	0	0	0	0	0
HEC 1	0	0	0	0	0	0
HEC 2	15	0	0	0	0	0
HEC 3	0	0	0	0	0	15

Thus, all the sparks ended up going to one core and got fizzled. We needed more sparks for the parallelisation to have an effect on the time taken.

For the largest puzzle, we see a decrease in the time taken to solve the puzzle, both while going from sequential to 2 cores, and while changing from 2 to 4 cores. On running these tests multiple times, we observe the same pattern, making us confident that there is a significant speedup observed with the 20 X 20 puzzle.

However, we also notice that the speed up is only about 63% from sequential to 4 cores, that is, the program runs in a little more than a third of the time, despite having 4 times the number of cores. This can be attributed to two main reasons. First, the algorithm is not completely parallelised, and has a lot of parts that must happen sequentially. This already limits the amount of speed up we can achieve according to Amdahl's Law. Secondly, our parallelisation strategy is simple, and creates too many sparks. On investigating this through threadscope, we find the following while running the 20 X 20 puzzle on 4 cores:



HEC	Total	Converted	Overflowed	Dud	GC'd	Fizzled
Total	487196	300	0	0	445365	41531
HEC 0	125474	57	0	0	119035	4938
HEC 1	120091	43	0	0	116269	3820
HEC 2	119295	67	0	0	105449	14116
HEC 3	122336	133	0	0	104612	18657

The program is creating way too many sparks, and most of them get garbage collected or fizzled. This could also be an additional factor as to why we don't see more of a speed up.

Further Work:

There are a few things to be done to further improve on as well as validate the results we have observed:

1. We should try running with even bigger puzzles to see if the pattern holds. We should see increased speedup, but this needs to be verified.
2. We should try running our code on lots of different puzzles of each size and average the results across them. This would help us account for puzzles that are "easier" or more "difficult", and further verify that our parallelisation approach is correct.
3. We should also work on depth limiting the parallelisation so we don't see as many sparks being garbage collected. We are not sure whether this will help the overall speed up and by how much, but it is definitely worth exploring more complicated parallelisation strategies.

Code:

```haskell
{-
    nonogram_solver.hs
    @author Manav Goel (mg3851) and Tanvi Hisaria (th2720)
-}

import System.Environment(getArgs, getProgName)
import Control.Monad (when, mplus, foldM)
import Control.Parallel (par, pseq)
import Data.IntMap (IntMap, insert, toList, fromList, (!))

data Color = White | Black
           deriving (Eq)
instance Show Color where
    show Black = "X"
    show White = "-"

type Nonogram = [[Color]]
type Hint = [Int]   -- represents a hint for a row or column e.g. [2, 4, 5]
data ColumnInfo = PlacedColor Color -- A single filled cell
                | BlackRun Int      -- Length of next black cell run
type ColumnInfoMap = IntMap [ColumnInfo]

-- Checks if a Column has any Black squares remaining
isColumnEmpty :: [ColumnInfo] -> Bool
isColumnEmpty [] = True
isColumnEmpty (PlacedColor Black : _) = False
isColumnEmpty (BlackRun _ : _) = False
isColumnEmpty (_ : xs) = isColumnEmpty xs

-- Tries consuming a color in a column, returns the correct column if valid
tryPlacingColor :: Color -> [ColumnInfo] -> Maybe [ColumnInfo]
tryPlacingColor White [] = Just []
tryPlacingColor Black [] = Nothing
-- Consume a matching PlacedColor
tryPlacingColor y (PlacedColor x : hs) = if x == y then Just hs else Nothing
tryPlacingColor White hs = Just hs
```

```haskell
-- Expand a BlackRun if we start one
tryPlacingColor Black (BlackRun n : hs) = Just $ replicate (n - 1) (PlacedColor
Black) ++ (PlacedColor White : hs)

-- Just grabs the correct column from the map to pass to tryPlacingColor
placeColorHelper :: Color -> ColumnInfoMap -> Int -> Maybe ColumnInfoMap
placeColorHelper colorToTry columnMap index = do
    hs <- tryPlacingColor colorToTry $ columnMap ! index
    return $ insert index hs columnMap

-- Main recursive driver for solving a puzzle
solve :: Int -> Int -> [Hint] -> ColumnInfoMap -> Maybe Nonogram
solve width columnIndex rowHints columnMap
        | null rowHints =
            if all isColumnEmpty (map snd $ toList columnMap)
                then return [[]]    -- All hints and ColumnInfo exhausted,
puzzle is solved
                else Nothing
        | null hint = do -- This specific hint has been exhausted, meaning the
row is complete
            updatedInfoMap <- foldM (placeColorHelper White) columnMap
[columnIndex .. width - 1]
            rows <- solve width 0 remainingHints updatedInfoMap    -- start
solving the next row
            return $ replicate (width - columnIndex) White : rows
        | otherwise -- Try to place a black and white square next in parallel,
keep the one that works
            = tryPlaceBlack `par` tryPlaceWhite `pseq` mplus tryPlaceBlack
tryPlaceWhite

        where
            (hint : remainingHints) = rowHints
            (h : hs) = hint

            tryPlaceBlack = do
                -- The current hint extends past the end of the puzzle
                when (columnIndex + h > width) Nothing
                -- Try to add h black cells to the row
                updatedInfoMap <- foldM (placeColorHelper Black) columnMap
[columnIndex .. columnIndex + h - 1]

                -- Try to place a white cell if we aren't at the end of the row
                im' <- if columnIndex + h == width
                        then return updatedInfoMap
                        else placeColorHelper White updatedInfoMap (columnIndex
+ h)

                -- Solve the rest of the row
                (row : rows) <- solve width (columnIndex + h + 1) (hs :
remainingHints) im'
                let row' = if columnIndex + h == width
                            then row
                            else White : row

                return $ (replicate h Black ++ row') : rows

            tryPlaceWhite = do
                when (columnIndex >= width) Nothing
                updatedInfoMap <- placeColorHelper White columnMap columnIndex

                -- Solve the rest of the row
                (row : rows) <- solve width (columnIndex + 1) rowHints
updatedInfoMap
                return $ ((White : row)) : rows

-- Helper that creates the ColumnInfoMap and passes it to solve
nonogram :: [[Hint]] -> Maybe Nonogram
nonogram [] = Nothing
nonogram [_] = Nothing
nonogram (_:_:_:_) = Nothing
nonogram [rows, columns] = solve (length columns) 0 rows myColumnMap
  where
    myColumnMap = fromList (zip [0 ..] $ map (map BlackRun) columns)

printNonogram :: Maybe Nonogram -> IO ()
printNonogram Nothing  = putStrLn "No solution!"
printNonogram (Just s) = mapM_ (putStrLn . concatMap show) s

strToInt :: String -> [[[Int]]]
strToInt a = read a::[[[Int]]]

main :: IO ()
main = do
    args <- getArgs
    case args of
        [filename] -> do
            contents <- readFile filename
            let rawPuzzle = strToInt (lines contents !! 0)
            printNonogram $ nonogram rawPuzzle
        _ -> do
            name <- getProgName
            putStrLn $ "Usage: " ++ name ++ "<filename>"
```

Output:

```
-----------------------------------
5 x 5 - 1 threads
XXXX
X--X
X--X
X-XX
XXX-


real	0m0.028s
user	0m0.000s
sys	0m0.031s


-----------------------------------
5 x 5 - 2 threads
XXXX
X--X
X--X
X-XX
XXX-


real	0m0.027s
user	0m0.016s
sys	0m0.000s


-----------------------------------
5 x 5 - 4 threads
XXXX
X--X
X--X
X-XX
XXX-
```

```
real	0m0.028s
user	0m0.000s
sys	0m0.031s


-----------------------------------
10 x 10 - 1 threads
XX-XXXXX--
--XXXX----
X-XXXXXX--
--X-XX----
----XXX---
--X-XX--X-
X-XXX-XXXX
X-X-XX----
XXX-XXXXXX
XXXXXX----


real	0m0.026s
user	0m0.000s
sys	0m0.016s


-----------------------------------
10 x 10 - 2 threads
XX-XXXXX--
--XXXX----
X-XXXXXX--
--X-XX----
----XXX---
--X-XX--X-
X-XXX-XXXX
X-X-XX----
XXX-XXXXXX
XXXXXX----


real	0m0.032s
```

```
user  0m0.000s
sys   0m0.031s


-----------------------------------
10 x 10 - 4 threads
XX-XXXXX--
--XXXX----
X-XXXXXX--
--X-XX----
----XXX---
--X-XX--X-
X-XXX-XXXX
X-X-XX----
XXX-XXXXXX
XXXXXX----


real  0m0.111s
user  0m0.031s
sys   0m0.031s


-----------------------------------
20 x 20 - 1 threads
----------XXX-------
---------XXXXX------
---------XXX-X------
---------XX--X------
------XXX-XXX-XXXX--
----XX--XX---XXXXXXX
--XXXXXX-X---X------
-XXXX---XX--XX------
--------X---X-------
-------XXX--X-------
-------XXXXXX-------
-XX---XXXXXXX-------
XXXXXX--XXX-X-------
X-XX--XX-X--X-------
```

```
---XXXX--X-X--XXX---
--------XXXX-XX-XX--
--------XXX--XXX-X--
-------XXX----XXX---
------XXX-----------
------XX-X----------


real  0m0.444s
user  0m0.375s
sys   0m0.047s


-----------------------------------
20 x 20 - 2 threads
----------XXX-------
---------XXXXX------
---------XXX-X------
---------XX--X------
------XXX-XXX-XXXX--
----XX--XX---XXXXXXX
--XXXXXX-X---X------
-XXXX---XX--XX------
--------X---X-------
-------XXX--X-------
-------XXXXXX-------
-XX---XXXXXXX-------
XXXXXX--XXX-X-------
X-XX--XX-X--X-------
---XXXX--X-X--XXX---
--------XXXX-XX-XX--
--------XXX--XXX-X--
-------XXX----XXX---
------XXX-----------
------XX-X----------


real  0m0.298s
user  0m0.453s
```

```
sys   0m0.078s


-----------------------------------
20 x 20 - 4 threads
----------XXX-------
---------XXXXX------
---------XXX-X------
---------XX--X------
------XXX-XXX-XXXX--
----XX--XX---XXXXXXX
--XXXXXX-X---X------
-XXXX---XX--XX------
--------X---X-------
-------XXX--X-------
-------XXXXXX-------
-XX---XXXXXXX-------
XXXXXX--XXX-X-------
X-XX--XX-X--X-------
---XXXX--X-X--XXX---
--------XXXX-XX-XX--
--------XXX--XXX-X--
-------XXX----XXX---
------XXX-----------
------XX-X----------


real  0m0.162s
user  0m0.500s
sys   0m0.063s
```