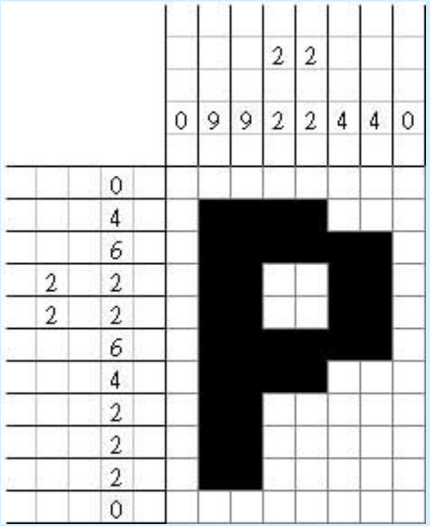
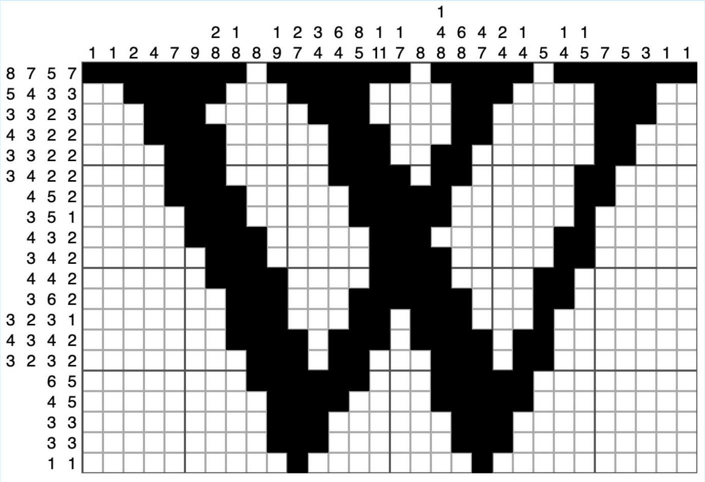
Nonogram Solver





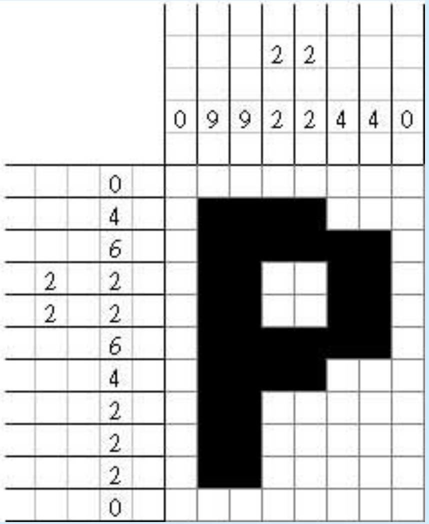
Parallel Functional Programming - Fall 2024

By: Dorothy Nelson, Jittisa (Jane) Kraprayoon

Index

Nonograms

- A **visual puzzle** to construct a picture by filling a grid of cells
- Constraints are provided for **rows** and **columns**
- Constraints specify how many blocks are consecutively filled. There must also be a space in-between the blocks.
- Initially every cell is unfilled, and players must make inferences.
- Chosen for their scalable complexity.



Data & Parsing

- We collected nonograms from a public github repo: https://github.com/mikix/nonogram-db
- We categorized puzzles by their sizes (e.g. 10x10 is small, 75x50 is large)
- `parseNonogram` was used to extract rowArgs and colArgs (row and column constraints) to be inputted into our algorithm

```
title "Bloop Bloop"
width 10
height 10

rows
3,2
3
2,2
4
4
2,2
4,1
4,2
2,2
1

columns
3
3,2
2,4
4
2
2
4
1,4
1,2,2
4

```

```
puzzles_db
  large
    brightly.txt
    kde.txt
    swing.txt
    tiger.txt
    wikimedia.txt
  medium
    16.txt
    21.txt
    42.txt
    100.txt
    101.txt
    102.txt
    blender.txt
    flower.txt
    gnome.txt
    rhino.txt
    spade.txt
    ubuntu.txt
  small
    bloop.txt
    dancer.txt
```

Algorithm

Our nonogram solver base algorithm can be described in three parts:

1) constraint satisfaction

2) iterative inference

3) backtracking for unresolved cases.

1) Constraint Satisfaction

The algorithm begins by iterating through each row and column constraint to compute possible placements of blocks for each line.

computeBlocksSeq :: Int -> [Int] -> [[Int]]

- takes the total line length and block constraints as inputs. It recursively places blocks into different start positions and continues with the remaining blocks.
- For example, given lineLength = 7 and lineConstraint = [2, 3], the function would output [[0, 3], [1, 4]], representing the start positions of the blocks.

generateBlocksSeq :: [[Int]] -> [Int] -> Int -> [[Int]]

- takes the output of computeBlocks—the potential starting positions of the blocks—and generates a binary array (Ints of 1s and 0s) to represent possible line configurations.

Possible placements are stored in PlacementsDict and updated at each iteration.

2) Iterative Inference

Purpose: Iteratively deduce definite cell values based on all possible line configurations.

- The main function is IterativeSolve which calls inferValues and updatePlacements. IterativeSolve recursively calls itself until the nonogram is solved.
- inferValues: If a cell is **filled** in all possible configurations, it must be **black**. If a cell is **unfilled** in all possible configurations, it must be **white**. -1 represents unknowns. Ex. [[0, 1, 0, 0], [0, 1, 1, 1]] → [[0, 1, -1, -1]]
- After inferValues, we run updatePlacements — prune the search space as some placements are now not possible

```
iterativeSolveSeq :: PartialSolution
                  -> PlacementsDict        -- Row and column placements
                  -> [Constraint]          -- Row constraints
                  -> [Constraint]          -- Column constraints
                  -> Set Int               -- Completed rows
                  -> Set Int               -- Completed columns
                  -> (PartialSolution, PlacementsDict, Set Int, Set Int)
```

3) Backtracking

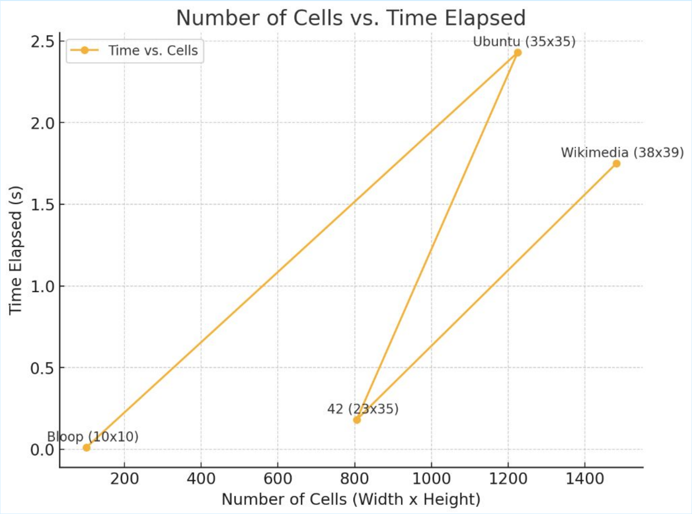
Purpose: **backtrack** Solve ambiguous nonograms with multiple solutions that cannot be resolved through iterative solving alone.

- Tries different placements within a row.
- Checks if the placement results in a **valid** grid.
- If it's valid, call backtrack recursively until all rows are completed. This is the base case of the recursion.

```
-- Output: a list of PartialSolution(s)
backtrack :: PartialSolution
          -> Array Int (Set [Int])  -- Row placements
          -> [Constraint]           -- Row constraints
          -> [Constraint]           -- Column constraints
          -> Set Int                -- Completed rows
          -> Set Int                -- Completed columns
          -> [PartialSolution]      -- Accumulated solutions
          -> [PartialSolution]      -- Final list of valid solutions
```

Sequential Benchmark

		Total Time Elapsed (s)
Small	Bloop (10x10)	0.01
Medium	Ubuntu (35x35)	2.43
	42 (23x35)	0.18
Large	Wikimedia (38x39)	1.75
	7 medium puzzles solved sequentially	36.77



Difficulty of nonogram comes not only from the size, but also how sparse the constraints are.

```
let filePaths = [ "puzzles_db/medium/42.txt", "puzzles_db/medium/blender.txt", "puzzles_db/medium/gnome.txt",
                  "puzzles_db/medium/spade.txt", "puzzles_db/medium/rhino.txt",
                  "puzzles_db/medium/ubuntu.txt", "puzzles_db/medium/flower.txt"]
```

Parallelization

Motivation for Parallelization

- Row and column processing can be done **independently**.
- Example: Computing starting placements for one row is unaffected by other rows.

Parallelization Strategy

- `Control.Parallel.Strategies` (`parMap, rdeepseq`). We parallelized specific functions contributing to the main algorithm: `inferValuesPar`, `computeBlocksPar`, and `generateBlocksPar`

Parallelization

We tested combinations of: `inferValuesPar`, `computeBlocksPar`, and `generateBlocksPar`

```
solveSequential :: FilePath -> IO ()
solveSequential = solveNonogramFromFile computeBlocksSeq generateBlocksSeq iterativeSolveSeq

solveParallelComputeBlocks :: FilePath -> IO ()
solveParallelComputeBlocks = solveNonogramFromFile computeBlocksPar generateBlocksSeq iterativeSolveSeq

solveParallelGenerateBlocks :: FilePath -> IO ()
solveParallelGenerateBlocks = solveNonogramFromFile computeBlocksSeq generateBlocksPar iterativeSolveSeq

solveParallelComputeGenerate :: FilePath -> IO ()
solveParallelComputeGenerate = solveNonogramFromFile computeBlocksPar generateBlocksPar
iterativeSolveSeq

solveParallelIterativeSolve :: FilePath -> IO ()
solveParallelIterativeSolve = solveNonogramFromFile computeBlocksSeq generateBlocksSeq iterativeSolvePar

solveFullyParallel :: FilePath -> IO ()
solveFullyParallel = solveNonogramFromFile computeBlocksPar generateBlocksPar iterativeSolvePar
```

INTRO NONOGRAMS ALGORITHM PARALLEL RESULTS NEXT STEPS

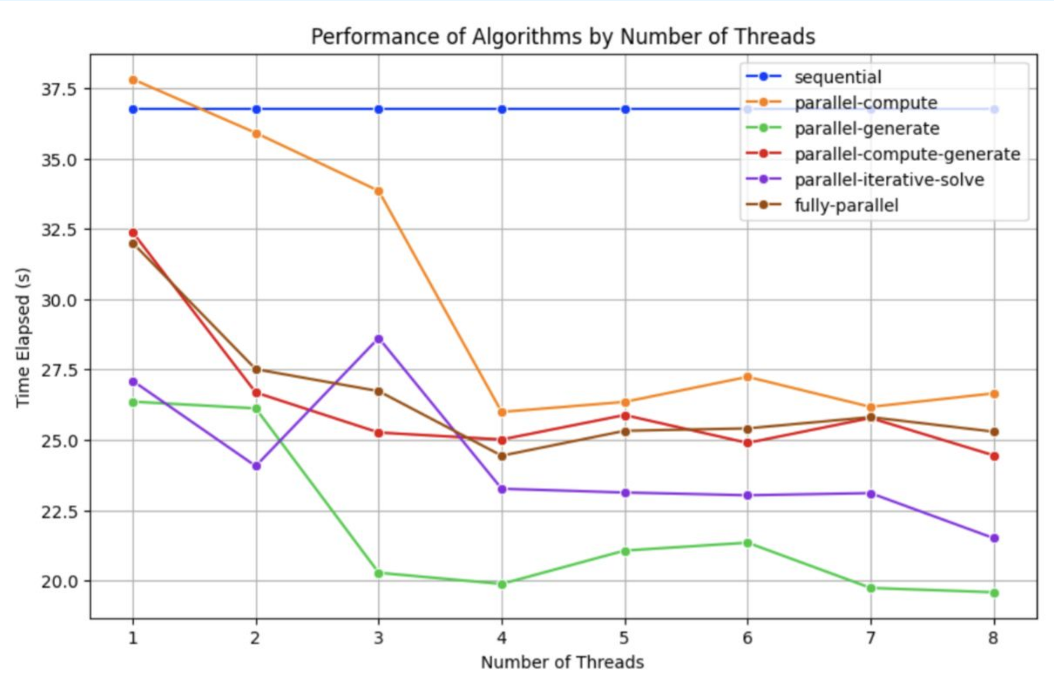
Results

	Max Speedup
Sequential	1.00
Parallel Compute	1.40
Parallel Generate	1.88
Parallel Compute Generate	1.50
Parallel Iterative Solve	1.71
Fully Parallel	1.50

Speedup = (Sequential benchmark) / (Shortest time elapsed for algorithm)

Parallel Generate: Achieved the **best performance** with a **1.88x speed-up**.

Results

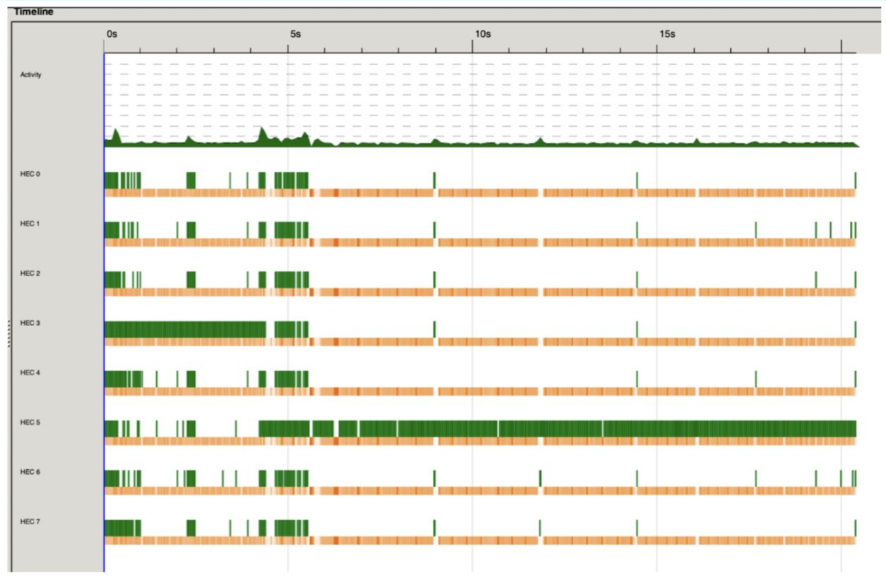


- The graph shows the **total time elapsed** with an increasing number of threads.
- For all **parallelized versions of the algorithm**, the elapsed time **decreased**, demonstrating **utilization of the threads**.
- All algorithms eventually **level out**, indicating **diminishing returns** with an excessive number of threads.

Next Steps

What are some future improvements?

Limitations from Iterative Solve

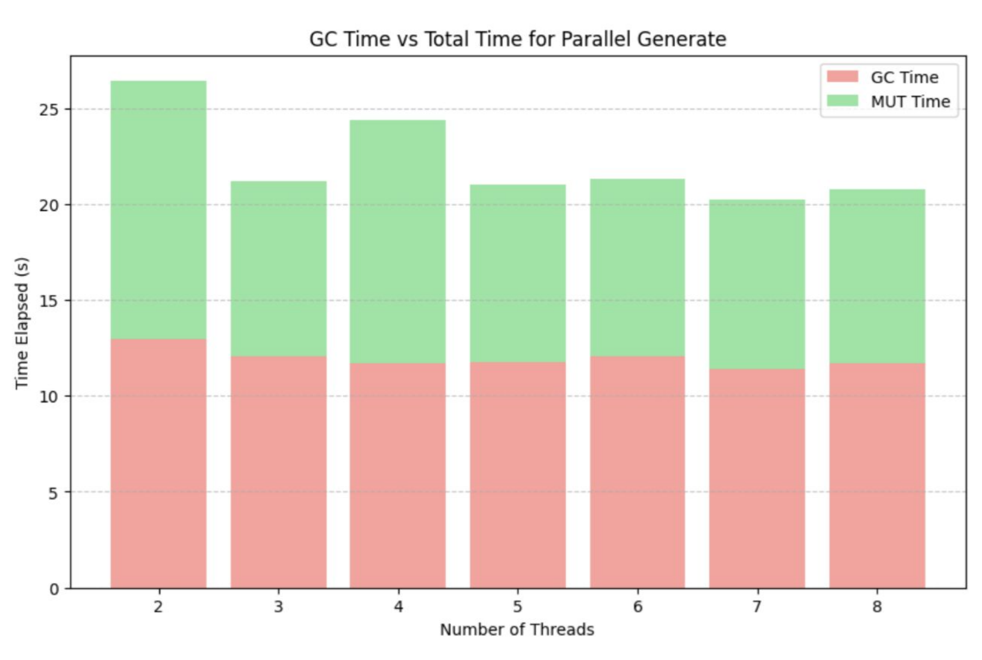


- Difficulty in parallelizing the main process of the algorithm — iterativeSolve.
  - The sequential nature of iterativeSolve makes parallelization challenging, as each step depends on the result of the previous one.
- The activity graph does show some success in parallelizing other parts of the algorithm: computeBlocksPar and generateBlocksPar

Limited use of backtracking

- The puzzles that we ended up testing on were **not ambiguous**, so **backtracking** was not utilized.
- **Future Improvements**:
  - Focus on parallelizing the **backtracking** part of the algorithm, as **multiple placements can be explored concurrently**.
  - Prioritize solving **smaller nonograms** primarily through backtracking.

GC Time



- Garbage collection (GC) takes up as much time as mutator operations
- Explore ways to **reduce GC time**, such as using **ParBuffer** to help manage memory usage.

Thank You!