COMSW4995 Project Proposal

Parallel Futoshiki Solver

Yao Wang (yw4589), Eric Cheng (hc3645)

1. Overview

The purpose of this project is to design and implement a parallel Futoshiki solver in Haskell, comparing the performance of a sequential solver with its parallel counterpart. The final program will take a text file describing a puzzle and return one or all valid solutions.

2. Puzzle Description

Futoshiki is a logic-based number puzzle similar to Sudoku, played on an $N \times N$ grid. The goal is to fill the grid with numbers from 1 to N such that:

1. Each row and column contains all numbers without repetition.
2. A set of given inequality constraints between cells ($e.g.\ (r1, c2) < (r1, c3)$) are satisfied.

These two simple rules lead to a search space that grows exponentially with N, making Futoshiki a good candidate for a parallel backtracking solver.

3. Algorithm

The base solver will use a recursive depth-first search (DFS) with constraint checking:

1. Select an unfilled cell.
    - A cell with the fewest valid candidates can reduce branching.
2. Generate valid candidates.

    For the chosen cell, collect all values $v \in [1..N]$ that:

    - Do not appear in the same row or column.

- Do not violate any inequality with already assigned neighbors.

3. Recurse.

   For each candidate value:

   - Temporarily assign it.
   - Recurse to solve the remaining grid.
   - If a contradiction occurs, backtrack.

4. Termination.

   The recursion stops when all cells are filled and all constraints hold. Each valid board is recorded as a solution.

This algorithm is a typical constraint satisfaction problem (CSP) solver. Its runtime depends heavily on branching factor and constraint density. For instance, a 6×6 Futoshiki with few inequalities may already require exploring thousands of partial boards.

4. Parallelization Plan

The Futoshiki solver naturally lends itself to task parallelism because each candidate assignment in the recursion tree can be explored independently.

We plan to parallelize the search using Haskell's $Control.Parallel.Strategies\ library$.

The main recursive function will map over all candidate assignments for the current cell, and each branch of the search tree will be evaluated in parallel using $parMap$ or a $parList$ strategy.

We will limit parallelization to the upper few levels of recursion. After the first few steps, the solver will switch to sequential mode for deeper branches.

5. Input and Output Plan

Input will come from a plain text file describing:

- Puzzle size N
- Preset cell values ($e.g.\ (0,1) = 4$)
- Inequality constraints ($e.g.\ (0,2) < (0,3)$)

Example:

```
None
size 5

initial:

(0,1)=3

(3,4)=2

constraints:

(0,0)<(0,1)

(1,2)>(2,2)

(2,3)<(2,4)
```

The program will include a small parser that reads this format into internal data structures.

Output:

- A printed grid for each valid solution, or
- A count of total valid solutions if requested via a command-line flag.

6. Evaluation and Testing

The project will measure:

- Runtime speedup of the parallel solver vs sequential baseline.
- Scalability across different core counts using +RTS -N2, -N4, etc.

- Impact of constraint density: puzzles with many inequalities should be solved faster due to reduced search space, while sparse ones provide a better stress test for parallelism.

Test puzzles will be created manually and, if possible, taken from open-source Futoshiki puzzle collections.

7. Expected Results

A correct, general Futoshiki solver that can handle at least N=6 or N=7. Parallel version achieving noticeable runtime improvement (2–3× expected on 4 cores). A short performance analysis and discussion in the final report, with ThreadScope visualization showing spark distribution and runtime behavior.