Analyzing Sequential vs. Parallel Computing Approaches for Futoshiki Puzzle Solutions

Abhishek Khatri
akhatri@uno.edu

Padam Jung Thapa
pthapa@uno.edu

Sourav Raxit
sraxit@uno.edu

Futoshiki Puzzle:

The Futoshiki puzzle, a logic-based numerical game, serves as an intriguing problem for exploring computational techniques in parallel and scientific computing. Translating to "inequality" in Japanese, Futoshiki combines the principles of Latin squares with additional constraints in the form of inequality symbols (e.g., ">" or "<") between adjacent cells. These constraints make the puzzle a compelling challenge for algorithmic problem-solving and optimization.

The puzzle is typically played on a $n \times n$ grid where the goal is to fill each cell with a number from 1 to n such that:

- Each number appears once in every row and column (like Sudoku).
- The inequality constraints between specific cells are satisfied.

Given its structured rules and combinatorial nature, solving Futoshiki puzzles becomes computationally intensive as the grid size increases. This makes it an ideal candidate for evaluating sequential and parallel computational approaches.

Sequential Approach:

In sequential approach, we have employed a traditional backtracking algorithm to solve the futoshiki puzzle. This involves systematically exploring possible solutions by filling one cell at a time, backtracking whenever a constraint is violated. While straightforward, this approach becomes inefficient for larger grids due to its exponential time complexity. The basic pseudocode being used to solve the puzzle is given below:

Pseudocode:

```
EMPTY_CELL = -1
NO_EMPTY_CELL = (-1, -1)
```

```
class Constraint:
  function Constructor(x1, y1, x2, y2):
    start = (x1, y1)
    end = (x2, y2)

  function isSatisfied(matrix):
    startValue = matrix.getValue(start.x, start.y)
    endValue = matrix.getValue(end.x, end.y)
    if startValue is EMPTY_CELL or endValue is EMPTY_CELL:
      return true
    return startValue > endValue

class Matrix:
  function Constructor(size):
    Initialize 2D array 'data' of size x size with EMPTY_CELL

  function setValue(row, col, value):
    Set data[row][col] to value

  function getValue(row, col):
    Return value at data[row][col]

  function isEmpty(row, col):
    Return true if data[row][col] is EMPTY_CELL, false otherwise

  function findEmptyCell():
    For each row in data:
      For each col in row:
```

```
            If cell at (row, col) is empty:
                Return (row, col)
        Return NO_EMPTY_CELL

    function isFull():
        Return true if findEmptyCell() is NO_EMPTY_CELL, false otherwise

class Solver:
    function Constructor(initialMatrix, constraints):
        matrix = initialMatrix
        constraints = constraints

    function doesSatisfyRules(row, col, value):
        Check if value is unique in row
        Check if value is unique in column
        For each constraint in constraints:
            Check if value satisfies the constraint
        Return true if all checks pass, false otherwise

    function solve():
        If matrix is full:
            Return true

        (row, col) = matrix.findEmptyCell()

        For value from 1 to matrix size:
            If doesSatisfyRules(row, col, value):
                matrix.setValue(row, col, value)
```

```
            If solve() is successful:
                Return true

            matrix.setValue(row, col, EMPTY_CELL)  // Backtrack

        Return false

    function solvePuzzle():
        Return result of solve()

function main():
    Read filename from command line arguments

    matrix = ReadMatrixFromFile(filename)
    constraints = ReadConstraintsFromFile(filename)

    Print initial matrix

    solver = Solver(matrix, constraints)

    start_time = current_time()
    solved = solver.solvePuzzle()
    end_time = current_time()

    Print "Duration: " + (end_time - start_time)

    If solved:
        Print "Puzzle solved successfully!"
        Print solved matrix
```

Else:

Print "No solution found."

Table 1.1: Execution time for sequential approach for futoshiki puzzle

Grid Size	1st Trial (µS)	2nd Trial (µS)	3rd Trial (µS)	4th Trial (µS)	5th Trial (µS)	6th Trial (µS)	Average (µS)
3*3	25	26	25	25	27	25	25.500
4*4	28	28	26	28	27	32	28.167
5*5	69	52	59	51	52	60	57.167
6*6	99	126	99	100	98	100	103.667
7*7	197	200	201	200	199	198	199.167
8*8	266	267	267	269	269	275	268.833
9*9	600	703	710	589	590	590	630.333
10*10	2363	2362	2370	2463	2348	2395	2383.500
11*11	13740	13560	13510	13596	13910	13864	13696.667
12*12	31524	33045	31686	30330	29252	29571	30901.33
13*13	53853	48826	48881	54270	52504	49366	51283.33
14*14	179101	163694	171184	162652	158784	161026	166073.50
15*15	1066239	1076347	1083707	1173863	1076398	1062886	1089906.67
16*16	4004203	4229001	4159062	4192158	4179283	4302343	4177675
17*17	38926833	40011021	39338582	38983721	40296064	39273399	39471603.33
18*18	43192337	43106604	43040419	43009233	43364223	43304238	43169509

Parallel Approach:

Here, in the parallel approach, we have tried solving the Futoshiki puzzle leveraging multi-threading capabilities provided by OpenMP to enhance computational efficiency. This method is particularly effective for larger grid sizes, where the complexity of the problem grows exponentially. By distributing tasks across multiple threads, the parallel approach aims to reduce execution time while maintaining the correctness of the solution.

The parallel approach for solving the Futoshiki puzzle is designed to efficiently handle its computational complexity by leveraging task-based parallelism using OpenMP. The process begins with Initialization and Input Handling, where the puzzle grid and constraints are read from an input file. The grid is stored as a 2D array, and constraints are represented as pairs of cell coordinates with inequality relationships. Memory allocation and data structures are initialized to support efficient processing. The core solving logic employs Parallel Recursive Backtracking, which uses OpenMP directives such as #pragma omp parallel and #pragma omp task to create parallel tasks for exploring value assignments in empty cells, while #pragma omp taskwait ensures synchronization at each recursion level. Constraint Checking ensures that each value assignment satisfies row and column uniqueness, inequality constraints, and cell emptiness, implemented through helper functions like canUseInRow, canUseInColumn, and isSatisfyConstraints. Once all cells are filled, Solution Validation confirms that the solution satisfies all constraints and maintains unique values in rows and columns. A shared atomic flag (solutionFound) is used to indicate when a valid solution is found, ensuring thread safety during updates. Finally, Execution Time Measurement is performed using high-resolution timers from the <chrono> library to evaluate performance improvements achieved through parallelization. This structured approach demonstrates significant reductions in execution time compared to sequential methods, particularly for larger grid sizes, showcasing the effectiveness of parallel computing in solving combinatorial problems like Futoshiki puzzles.

The parallel approach significantly reduces execution time compared to sequential methods, particularly for larger grid sizes. By dividing the computational workload among multiple threads, it minimizes idle time and accelerates the exploration of potential solutions. Additionally, OpenMP's task-based model allows dynamic load balancing, ensuring efficient utilization of available processing resources.

Pseudocode:

```
Class FutoshikiSolver:
  // Class members
  matrix: 2D integer array
  size: integer
  threshold: integer
  solutionFound: boolean
  executionTime: time duration
  constraints: list of pairs of cell coordinates

  Method Initialize(filename):
    Open file named filename
    Read size from file
    Create matrix with dimensions size x size
    For each row and column in matrix:
      Read value from file and store in matrix
    While file has more lines:
      Read constraint from file
      Add constraint to constraints list
    Close file

  Method Cleanup:
    Free memory allocated for matrix

  Method Solve:
    Start timer
    Begin parallel region
      Create single task:
```

```
        Call SolveRecursive
    End parallel region
    Stop timer
    Calculate executionTime

Method SolveRecursive:
    emptyCell = FindEmptyCell
    If no empty cell found:
        If PuzzleIsSolved:
            Set solutionFound to true (ensure this is atomic)
        Return

    For value from 1 to size:
        If RulesAreSatisfied for emptyCell and value:
            Place value in emptyCell
            Create new parallel task:
                Call SolveRecursive
            Wait for all child tasks to complete
            If solution not found:
                Remove value from emptyCell (backtrack)

Method RulesAreSatisfied(row, column, value):
    Return (CanUseInRow AND CanUseInColumn AND
            ConstraintsSatisfied AND CellIsEmpty)

Method CanUseInRow(row, value):
    For each cell in row:
        If cell contains value:
            Return false
```

```
  Return true

Method CanUseInColumn(column, value):
  For each cell in column:
    If cell contains value:
      Return false
  Return true

Method ConstraintsSatisfied(row, column, value):
  For each constraint in constraints:
    If constraint involves cell at (row, column):
      Check if value satisfies the constraint
      If not satisfied:
        Return false
  Return true

Method CellIsEmpty(row, column):
  Return true if matrix[row][column] is -1, false otherwise

Method PuzzleIsSolved:
  Check if all constraints are satisfied
  Check if all rows and columns have unique values
  Return true if both conditions are met, false otherwise

Method FindEmptyCell:
  For each row and column in matrix:
    If matrix[row][column] is -1:
      Return (row, column)
  Return (-1, -1) to indicate no empty cell found
```

```
  Method PrintMatrix:
    For each row in matrix:
      For each column in matrix:
        Print value at matrix[row][column]
      Print new line

Main Program:
  Try:
    Create FutoshikiSolver object
    Initialize solver with input file
    Print "Initial puzzle:"
    Call PrintMatrix
    Call Solve
    Print "Solved puzzle:"
    Call PrintMatrix
    Print execution time
  Catch any errors:
    Print error message
```

Table 1.2: Execution time for parallel approach for futoshiki puzzle

Grid Size	1st Trial (μS)	2nd Trial (μS)	3rd Trial (μS)	4th Trial (μS)	5th Trial (μS)	6th Trial (μS)	Average (μS)
9*9	1127	1017	1181	1278	871	1055	1088.17
10*10	1393	2710	1588	2062	2377	2541	2111.83
11*11	4025	3495	8941	9244	3859	5774	5889.67
12*12	86864	138383	90654	88007	49431	87608	90157.83
13*13	9586	10969	10900	14982	9679	16396	12085.33
14*14	47024	44585	43180	31580	47774	30471	40769
15*15	257184	179477	173097	180050	176385	258639	204138.67
16*16	839515	753703	696370	729869	680934	883445	763972.67
17*17	6252646	6405129	6300495	6354036	6968059	6367269	6441272.33
18*18	7466500	7476412	7712529	7162431	7274402	7160660	7375489

Comparision between the execution times of sequential approach and parallel approach:

Two approaches were implemented to solve the Futoshiki puzzle: a sequential method using a backtracking algorithm without multithreading, and a parallel method utilizing OpenMP with the backtracking algorithm. The study examined grid sizes from 3x3 to 18x18, with the parallel approach focusing on larger grids starting from 9x9.

Execution time, measured in microseconds, served as the performance metric for both approaches. A consistent pattern emerged: as grid size increased, so did the execution time.The sequential approach demonstrated a clear upward trend in execution time as grid sizes grew:

- Smaller grids (3x3 to 8x8) were solved in less than 300 microseconds.
- Mid-sized grids (9x9 to 13x13) required between 630 and 51,283 microseconds.

- Larger grids (14x14 to 18x18) saw a dramatic increase, with the 18x18 grid taking up to 43,169,509 microseconds.

The parallel approach, tested on grids from 9x9 to 18x18, showed the following performance:

- 9x9 to 11x11 grids were solved in 1,088 to 5,889 microseconds.
- 12x12 to 16x16 grids took between 12,085 and 763,972 microseconds.
- The most complex grids (17x17 and 18x18) required 6,441,272 and 7,375,489 microseconds respectively.

Compilation of codes:

System Specifications:

ASUS TUF Gaming F15 16 GB RAM, 256 GB SSD, Core i5 10th generation

Software Used:

Microsoft Visual Studio for coding

MingW64 for C++ compiler

For sequential approach:

g++ -o futoshiki_puzzle constraint.cpp fileio.cpp matrix.cpp solver.cpp main.cpp

./futoshiki_puzzle.exe input_3x3.txt

For parallel approach:

g++ -o futoshiki_puzzle futoshiki.cpp main.cpp -fopenmp

./futoshiki_puzzle.exe input_9x9.txt

Comparative Analysis:

To better illustrate the performance difference between the sequential and parallel approaches for solving Futoshiki puzzles, we can compare the execution times for grid sizes where data is available for both methods:

Table 1.3: Compartive analysis of the execution time between two approaches:

Grid Size	Sequential (μs)	Parallel (μs)	Speedup Factor
9x9	630.333	1,088.17	0.58x
10x10	2,383.500	2,111.83	1.13x
11x11	13,696.667	5,889.67	2.33x
12x12	30,901.33	90,157.83	0.34x
13x13	51,283.33	12,085.33	4.24x
14x14	166,073.50	40,769.00	4.07x
15x15	1,089,906.67	204,138.67	5.34x
16x16	4,177,675.00	763,972.67	5.47x
17x17	39,471,603.33	6,441,272.33	6.13x
18x18	43,169,509.00	7,375,489.00	5.85x

Key Observations:

1. **Performance Crossover:**

   The parallel approach's effectiveness becomes evident as grid sizes increase. For smaller grids, the sequential method outperforms the parallel one. However, as grid dimensions grow, the parallel approach demonstrates superior performance, as illustrated in the comparison table.

2. **Scalability:**

   The parallel approach exhibits improved scalability with increasing grid sizes. For the largest grids tested, the parallel method executes approximately five times faster than its sequential counterpart, highlighting its efficiency in handling complex puzzles.

3. **Overhead for smaller grids:**

When dealing with smaller grid dimensions, the parallel approach shows no significant performance improvement over the sequential method. This is likely attributed to the overhead associated with thread creation and management in parallel processing, which outweighs potential benefits for simpler puzzles.

4. **Peak Efficiency:**

   The parallel approach achieves its highest efficiency for grid sizes between 15x15 and 17x17, demonstrating speedup factors exceeding 5x. This range represents the sweet spot where parallelization benefits are maximized relative to computational complexity.

5. **Anomalies:**

   Some unexpected results are observed in the data, particularly for the 12x12 grid, where the parallel approach unexpectedly underperforms. These anomalies could stem from various factors, including thread contention issues or the specific complexity of the puzzle configuration (matrix and constraints) used in the test cases.

Conclusion:

OpenMP's parallel approach significantly enhances the solving speed of large Futoshiki puzzles, demonstrating its effectiveness as a robust tool for complex grids. However, the performance gains are not uniform across all puzzle sizes. The choice between sequential and parallel methods should be made thoughtfully, considering the specific dimensions of the grid. For smaller puzzles, the overhead of parallelization might outweigh the benefits, while larger grids are more likely to see substantial speedups from the parallel approach.

Future work:

Future research directions for enhancing the Futoshiki puzzle solver could include:

Algorithm Optimization

Refining the parallel algorithm to perform efficiently across a broader spectrum of grid sizes, with a particular focus on addressing the performance discrepancies observed in mid-range grids such as the 12x12 puzzle.

Anomaly Investigation

Conducting a thorough analysis of the unexpected performance dips, especially for the 12x12 grid, to identify and resolve potential design flaws or inefficiencies in the parallel implementation. This debugging process could reveal insights that lead to overall improvements in the algorithm's robustness.

CUDA Implementation

Exploring the potential of GPU acceleration by developing a CUDA-based version of the Futoshiki solver. This approach could potentially offer even greater performance gains, particularly for larger grid sizes. A comparative study of execution times between CPU-based parallel processing and GPU-accelerated solutions would provide valuable insights into the most effective solving strategies for different puzzle complexities.By pursuing these avenues of research, we could potentially achieve more consistent and superior performance across all grid sizes, further optimizing the solution for Futoshiki puzzles.

References:

David Swarbrick's Futoshiki Solver: *A GitHub repository containing a Python script to solve Futoshiki puzzles*

Haraguchi, Kazuya. (2013). *The Number of Inequality Signs in the Design of Futoshiki Puzzle*. JIP. 21. 26-32. 10.2197/ipsjjip.21.26.