COMS W4995 003
Parallel Functional Programming
Fall 2024

Instructor: Prof. Stephen A Edwards

Student Name: Milin Saini

Student UNI: mks2249

TITLE: GENERALIZED TIC TAC TOE SOLVER

Problem Definition

Generalized Tic Tac Toe, extending the classic 3x3 grid to larger sizes (N×N), introduces increased computational complexity, rendering the optimal move determination an NP-complete problem. This project explores the application of parallel functional programming in Haskell to enhance the performance of solving NxN Tic Tac Toe. By implementing both sequential and parallel versions of the Minimax algorithm, we aim to evaluate the performance gains achievable through parallelization across different board sizes (3x3, 4x4, 5x5).

Game Rules and Objectives

Game Rules**:**

- **Board Configuration:** The game is played on an N×N grid, where N ∈ {3, 4, 5}.
- **Players:** Two players (X and O) take alternate turns, placing their symbols on empty cells.
- **Win Condition:** Achieving K consecutive symbols in a horizontal, vertical, or diagonal line, where K = N.
- **Draw Condition:** All cells filled without any player meeting the win condition.

Objectives:

- Implement the Minimax algorithm to determine the optimal move for a given game state.
- Leverage Haskell's parallel capabilities to evaluate initial moves concurrently, enhancing solver performance.
- Measure and compare execution times of sequential and parallel solvers across varying board sizes.
- Provide insights into the scalability and efficiency of parallel functional programming in solving NP-complete problems by measuring the execution time and looking at the sparks statistics to determine the performance of the parallelized solver compared to sequential solver.

Development Environment

The project was developed using GHC 9.6.6., and the latest version of Stack 3.1.1. The project was setup on a Windows system. GHCup was used to seamlessly download and install Haskell and dependencies to set up Threadscope. The processor details that determine parallel capabilities of the system are shown in the table below. Further instructions to run the project and get results can be found in the README file.

Brand	Intel
Model	i5-8265U
Cores	4
Hardware Threads	8

Methodology

We implemented a simple minimax solver without alpha-beta pruning as the baseline sequential solver. Minimax recursively explores possible moves, simulates outcomes, and tries to pick the optimal move. For this project, we focus on a two-player scenario where PlayerX attempts to maximize the outcome, and PlayerO attempts to minimize it. However, due to reasons explained in the challenges section , we added to alpha beta pruning to the algorithm for further optimization.

Sequential Minimax Algorithm

A straightforward minimax algorithm that: - Checks for terminal states (win/loss/draw). - Recursively explores all available moves and chooses the best outcome for the current player.

Parallel Solver

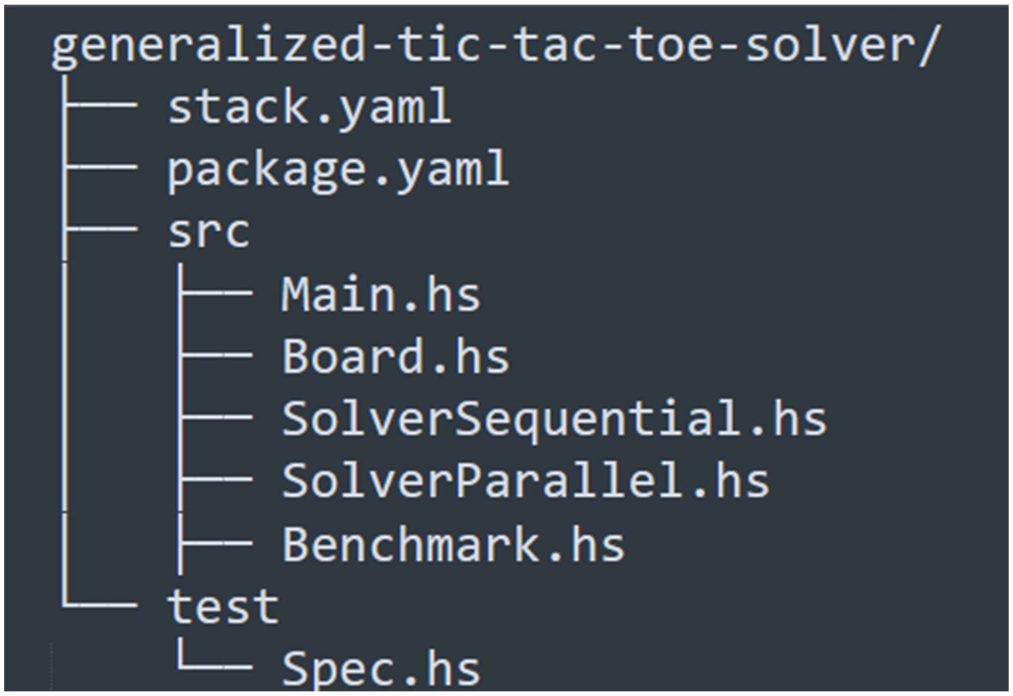
As the branching factor increases (larger boards or more moves), the computation time grows significantly. To improve performance, we parallelize the initial move evaluations using Haskell's `parallel` and `deepseq` libraries. On the top-level of the minimax tree (the set of possible next moves), we use `parList rdeepseq` to evaluate each possible move's resulting score in parallel. - This allows us to utilize multiple cores and potentially reduce the runtime for larger boards.

Benchmarking

We measure performance on board sizes 3×3, 4×4, and 5×5. Profiling with grid sizes 4x4, 5x5 posed great challenge in terms of runtime duration. Due to time constraints, benchmarking performance for sizes 4x4, 5x5 was dropped. We also added timing functions in the code for performance evaluation and built the project to enable profiling by Threadscope.

Code Structure

The code structure and description is given as follows:


```
generalized-tic-tac-toe-solver/
├── stack.yaml
├── package.yaml
├── src
│   ├── Main.hs
│   ├── Board.hs
│   ├── SolverSequential.hs
│   ├── SolverParallel.hs
│   ├── Benchmark.hs
└── test
    └── Spec.hs
```

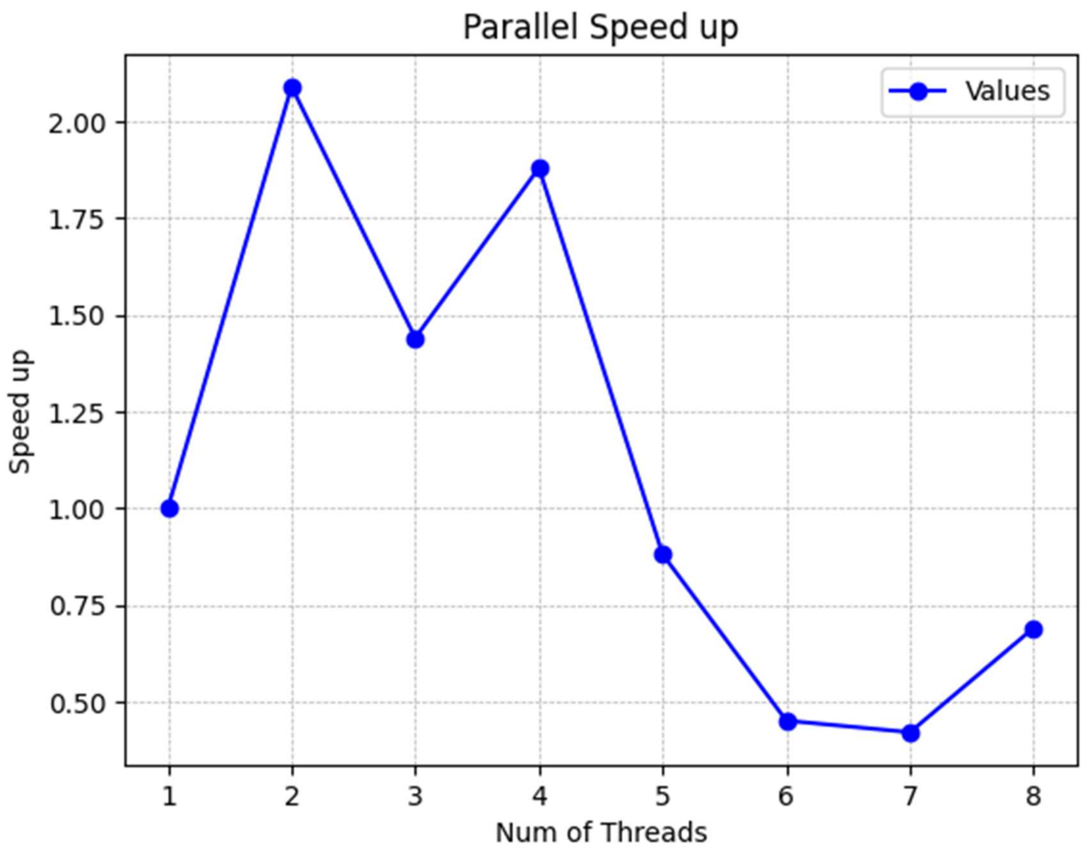
- stack.yaml: Stack configuration file with necessary resolver and extra-deps.
- package.yaml: Package configuration listing dependencies, executables, test suites, etc.
- src/Main.hs: Entry point of the program, sets up benchmarking scenarios, runs sequential and parallel solvers.
- src/Board.hs: Defines board types, players, moves, and related functions.
- src/SolverSequential.hs: Implements the sequential minimax solver.
- src/SolverParallel.hs: Implements the parallel minimax solver using Haskell's parallel strategies.
- test/Spec.hs: Basic test cases for correctness on small boards.
- src/Benchmark.hs: Code for running timing tests and reporting performance metrics.

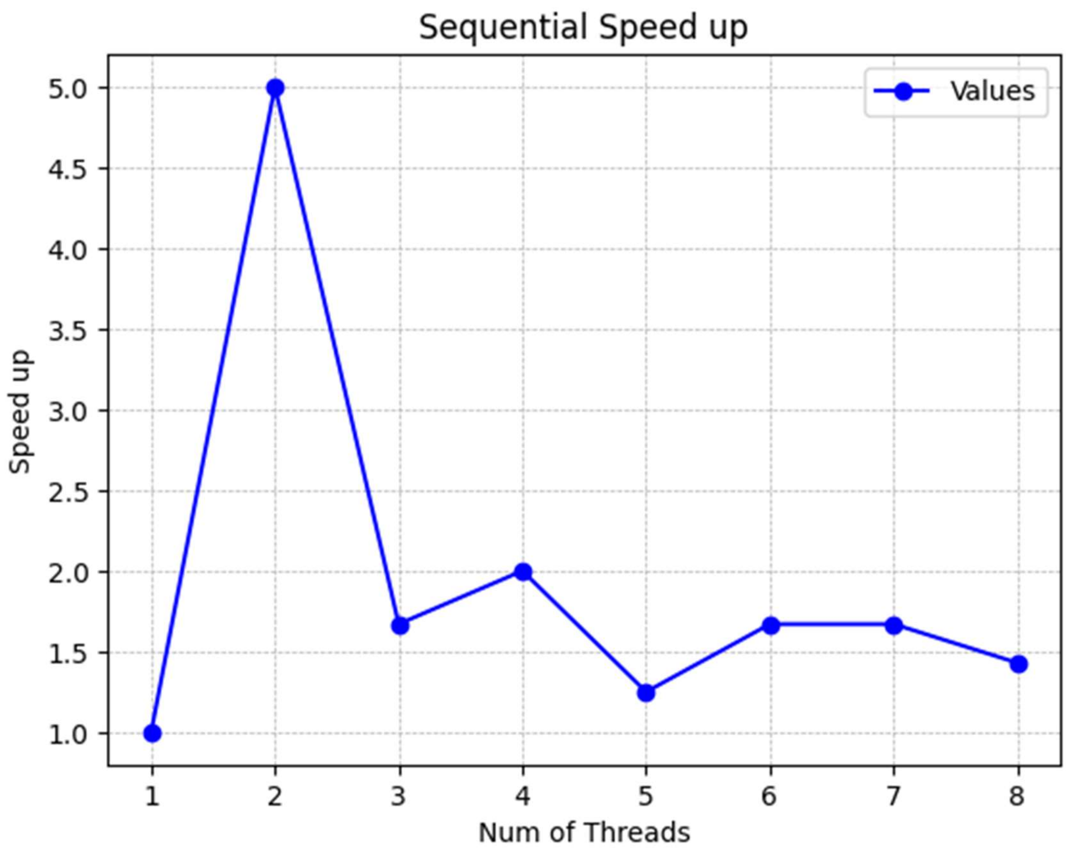
Results

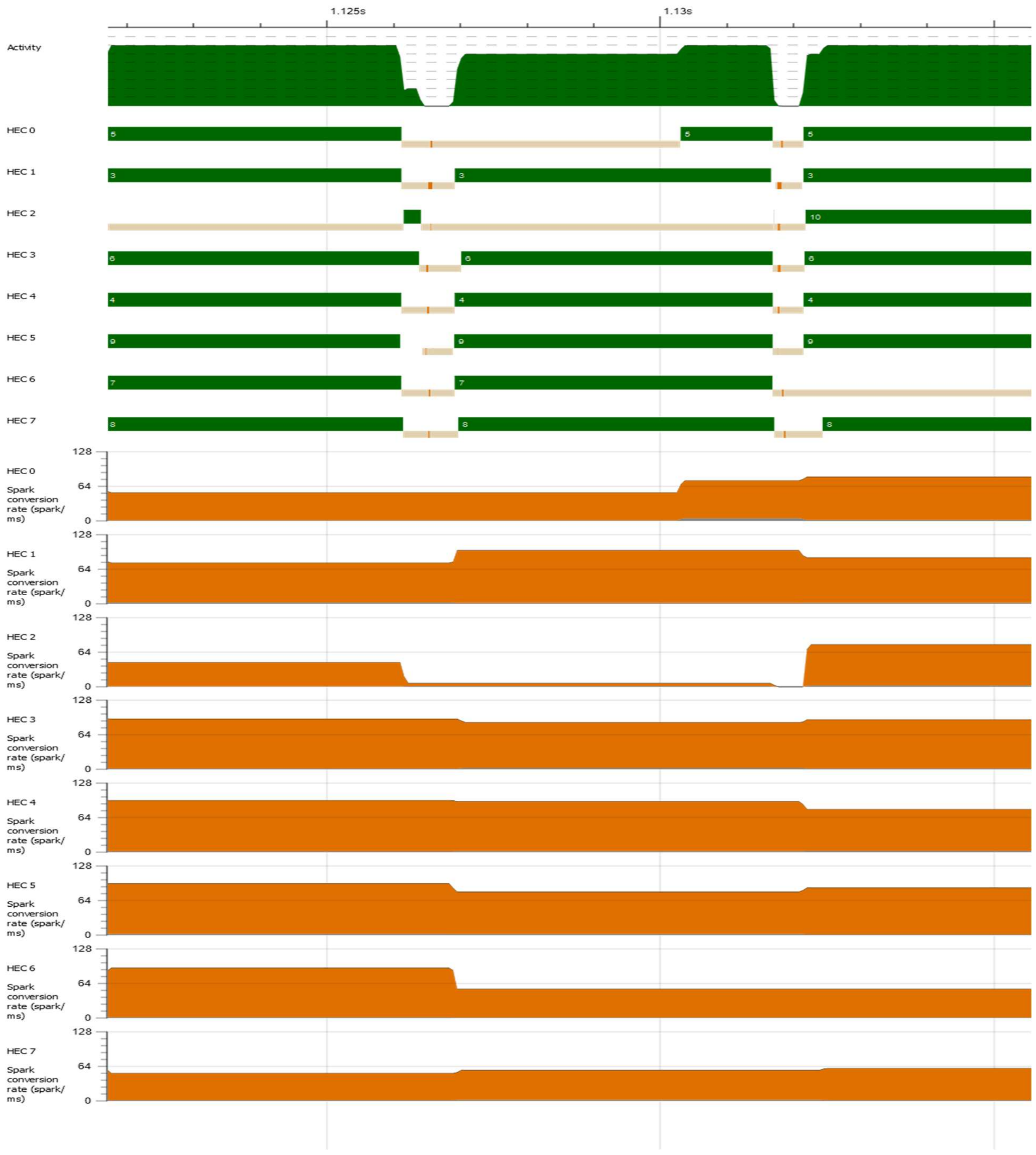
The total execution time (sequential and parallel), the sequential execution time and parallel execution time were recorded as part of the experiments. The speed up is also recorded for these cases. The results show that there is room for improving the performance of the parallelized algorithm. Also, the spark statistics were captured through Threadscope log files. The spark statistics indicate that the parallelization part of the program needs to be optimized as too many sparks are being generated with very little conversion. All the results are shown in the tables and figures below:

Number of threads	Total Execution Time (in sec)	Combined Speed up	Sequential(ms)	Seq Speed up	Parallel (ms)	Parallel Speed up
1	3.614	1.00	156.25	1.00	3140.63	1.00
2	2.135	1.69	31.25	5.00	1500.00	2.09
3	1.971	1.83	93.75	1.67	2187.50	1.44
4	1.816	1.99	78.13	2.00	1671.88	1.88

5	1.778	2.03	125.00	1.25	3562.50	0.88
6	1.709	2.11	93.75	1.67	6906.25	0.45
7	1.647	2.19	93.75	1.67	7421.88	0.42
8	1.697	2.13	109.38	1.43	4562.50	0.69


Parallel Speed up
Values
Speed up
2.00
1.75
1.50
1.25
1.00
0.75
0.50
1
2
3
4
5
6
7
8
Num of Threads


Sequential Speed up
Values
Speed up
Num of Threads
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0
1
2
3
4
5
6
7
8



Challenges

Few significant challenges were encountered during the development of this project. They are listed as follows:

- Expensive computations with increasing Grid Size : Considering the grid size 4x4, the standard Minimax algorithm will evaluate moves and perform calculations to the order of trillions. Even on a powerful processor such as Apple Pro M1 chip, such a program will take several days to run. As a result, alpha-beta pruning was added to the solver logic to optimize performance further. However, this still results in very high execution time. Therefore, for the purposes of the project, the evaluation of the algorithms for 4x4 and 5x5 grids was forfeited and only the 3x3 case is considered.

- Anomalous speed up seen in sequential solver execution time. When N=2 threads are used, a sharp speed up is seen in the sequential solver performance. The cause is unknown and further investigation is required.
- Execution times is very high in parallelized version of the algorithm. Even with the addition of alpha beta pruning, the execution time of the parallel code is significantly higher than the sequential version. Even though the speed up with increasing number of hardware threads follows the expected pattern, the performance is not good. Excessive parallelization and granularity in the code are suspected to be causing these results. The parallel algorithm needs to be optimized further, by either increasing chunk sizes or removing redundant nested parallelization. Due to time constraints, the optimization remains pending.

Future Work

The scope for improving the performance of the Parallel Solver is vast. One approach to improve the performance would be to take care of excessive parallelization in the code. There is parallelization at the first level in `bestMoveParallel` with parList but also in `maximizeAB` and `minimizeAB` with nested parList calls. Too much parallelization leads to generation of too many sparks resulting in excessive overhead.

Another factor to consider is the size of the grid. As 3x3 grid is a small problem, there may not be significant difference in performance with respect to the sequential program. Overheads in parallelization outweigh the processing gain in small grid size.

Finally, we can consider using other techniques such as Position Memoization and Early Gane Tree Truncation, to speed up and improve the sequential program itself, so that it is easier to profile in larger grid sizes.

References

- Alpha Beta Pruning, https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning
- Minimax, https://en.wikipedia.org/wiki/Minimax
- GHCup, https://www.haskell.org/ghcup/
- Hackage, https://hackage.haskell.org/
- Threadscope Tour, https://wiki.haskell.org/index.php?title=ThreadScope_Tour