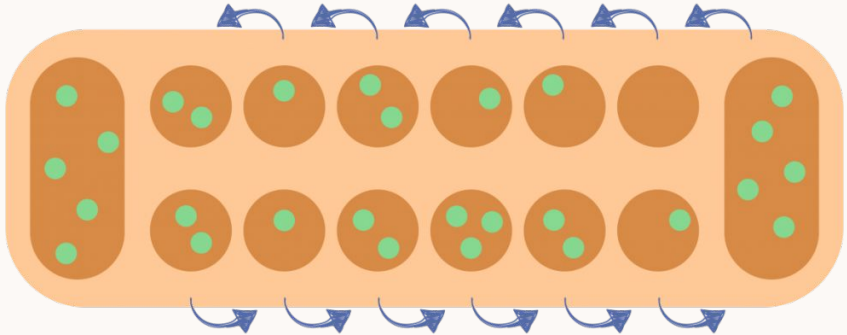
Fall 2024

Parallel Functional Programming

Mancala Parallelization with Minimax: A Functional Approach



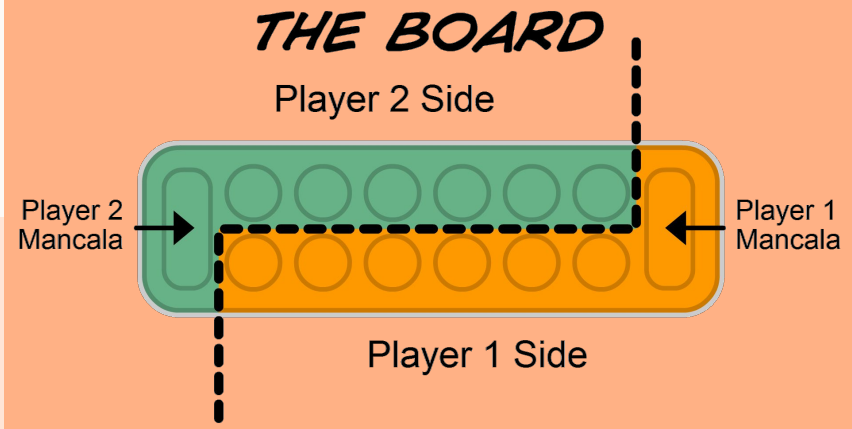
Daniel Manjarrez | Caiwu Chen | Sindhu Krishnamurthy

Introduction

Intro

Rules of the Game

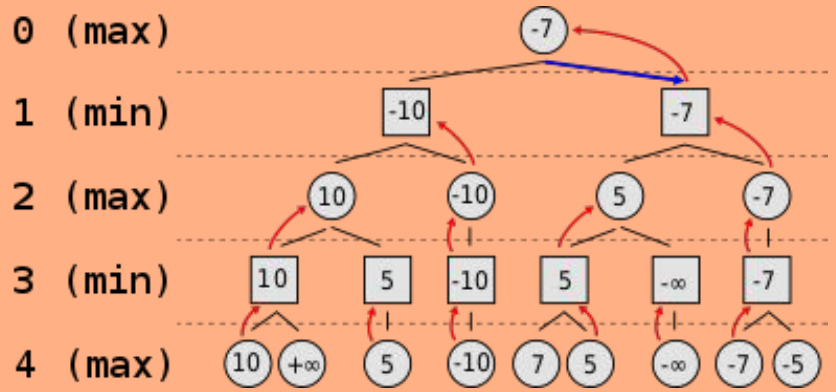
- Board Layout
  - 12 small pits filled with seeds and 2 stores for captured seeds
- Player Actions
  - Choose seeds from pits on their side of the board
  - Deposit the seeds one by one counterclockwise around the board
  - Only deposit seeds in player's own store
  - If a seed lands on an empty pit on the player's side, they capture the seeds in the pit opposite it
  - If the last seed lands in the player's store, they get an extra run
  - After a player distributes all seeds from a pit, the next player takes their round
- Objective
  - End game if all pits on one side of the of the board is empty
  - Win by collecting the most seeds in your store



With Alpha-Beta Pruning

Minimax Algorithm

- Recursively explore possible game states, switching between MaximizingPlayer and MinimizingPlayer
- Evaluate each state using heuristic function
- Scores are propagated up the tree
  - Max nodes select move with max score
  - Min nodes select move with min score
- Best move chosen based on max score at root node
- Tracks Alpha (MaximizingPlayer's best score) and Beta (MinimizingPlayer's best score).
- Prunes branches when a move's score is worse than the current Alpha or Beta.



Details of Implementation

- `Usage: Parallel-Minimax-Mancala <depth> <parallelDepth>`
- The game state begin with 12 pits and each pit contain 4 seeds.
- Minimax use simple heuristic for board evaluation.
  - For Player1: How many more seeds are in Player1's store compared to Player2's store?
  - For Player2: How many more seeds are in Player2's store compared to Player1's store?
- Alpha set to negative infinity while Beta sets to positive infinity.

Approach

Initial Approaches

Sequential -> Parallel Minimax

Sequential Minimax with Alpha-Beta pruning, then parallel Minimax past a given threshold

- Prune early, when it has most impact!

A

Parallel -> Sequential Minimax

Parallel Minimax, then sequential Minimax with Alpha-Beta pruning, past a given threshold

- Parallelize at top levels with higher branching factor

B

Workload-Based Parallel Minimax

Parallel Minimax when number of valid moves is past a certain threshold; otherwise, sequential Minimax with Alpha-Beta pruning

- Only parallelize larger workloads

C

Tested Approaches

How do we parallelize the Minimax Algorithm?

A

Sequential Minimax with Alpha-Beta pruning, then parallel Minimax past a given threshold

B

Parallel Minimax, then sequential Minimax with Alpha-Beta pruning, past a given threshold

C

Parallel Minimax when number of valid moves is past a certain threshold; otherwise, sequential Minimax with Alpha-Beta pruning

D

Hybrid Minimax: Sequential Minimax for first node at each level, Parallel Minimax for sibling nodes

D

Young Brothers Wait Concept

- Wait until leftmost node is evaluated first sequentially with alpha-beta pruning
- Then, remaining nodes are evaluated in parallel using the narrowed bounds
- Lowers overhead from parallelization by still enabling some extent of pruning

Chosen Implementation

Minimax Function

- When depth>=parallelDepth
  - Process first valid move sequentially using SeqMinimax to establish initial alpha-beta bounds
  - Use bounds to process remaining moves in parallel
    - Recursively call Minimax with parMap and rdeepseq
  - Combine all results and return best value
- Otherwise
  - Evaluate all moves sequentially using SeqMinimax with alpha-beta pruning
  - Select and return best value

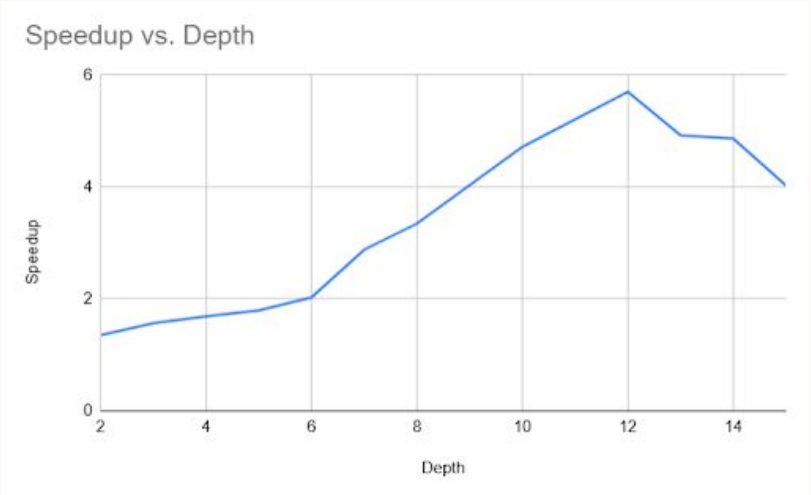
SeqMinimax Function

- For each move, recursively call SeqMinimax with new game state and switched player to get value
- If maximizingPlayer
  - Update alpha bound with value if bigger
- If minimizingPlayer
  - Update beta bound with value if smaller
- If alpha >= beta, return the updated value (and thereby stop exploring the branch)
- Otherwise, continue exploring remaining moves in current branch

Performance Evaluation

Speedup vs. Depth

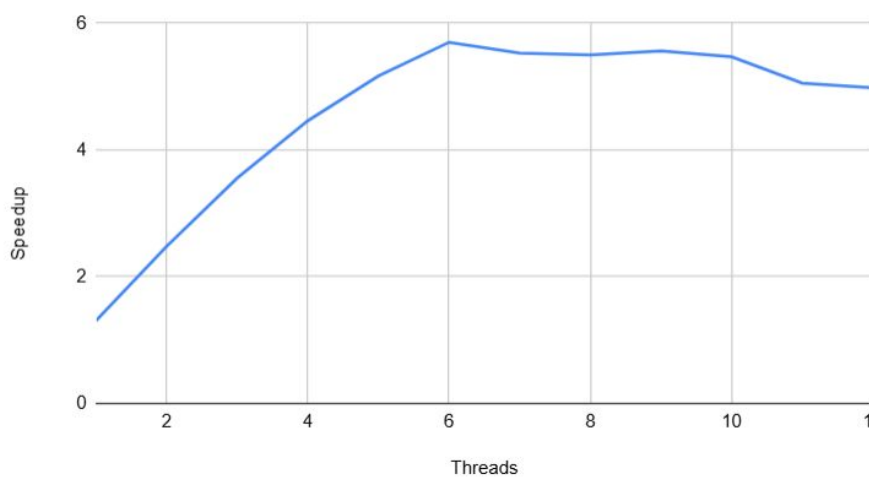
- As the Depth of the Search Tree grew so did the *speedup*
- The increase kept up until it peaked at a Level 12 Search Tree which then the *speedup* began to slowly decrease as the Search Tree continued to grow
- Overall, all the different levels of Search Trees we test benefitted from a *speedup* as a result of the parallelization performed



Speedup vs. Threads

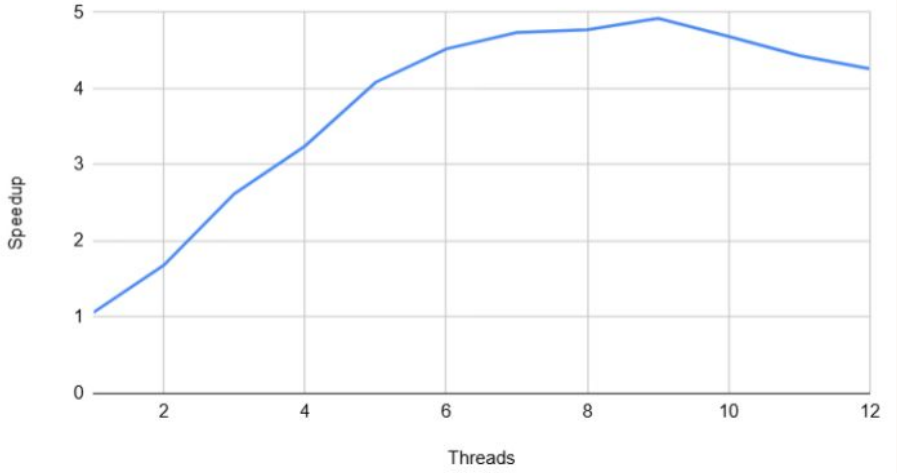
A

Smaller Search Trees <= Depth 12



B

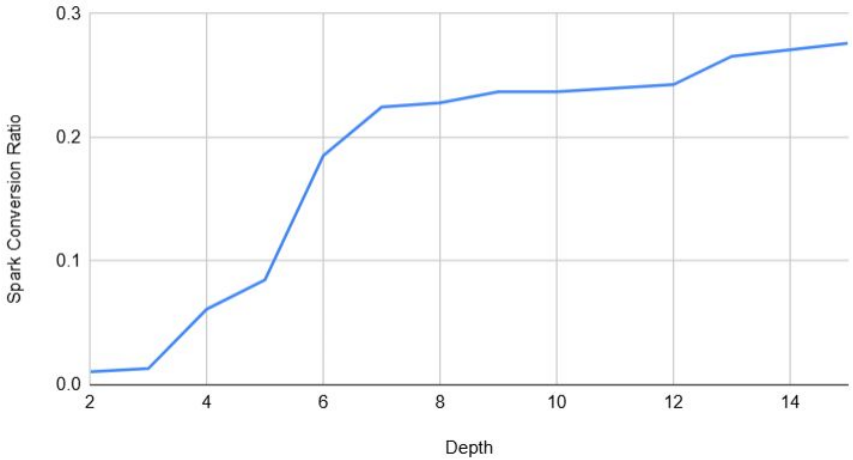
Larger Search Trees > Depth 12



Spark Conversion Ratio

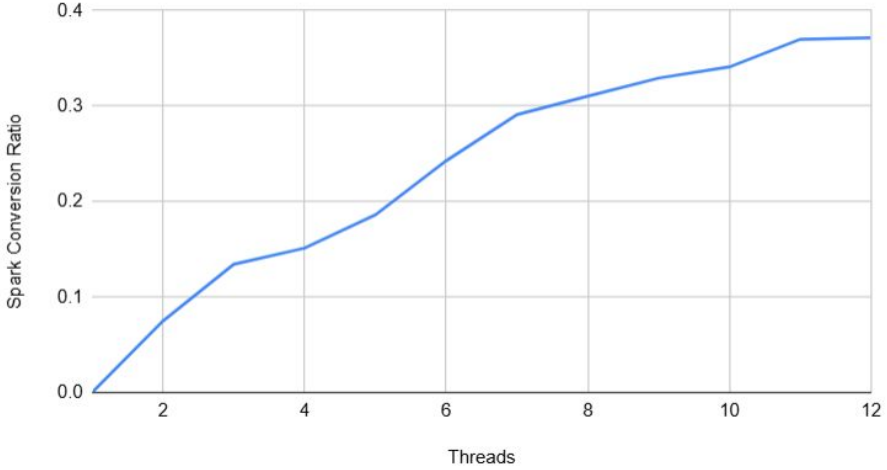
A

Spark Conversion Ratio vs. Search Tree Depth



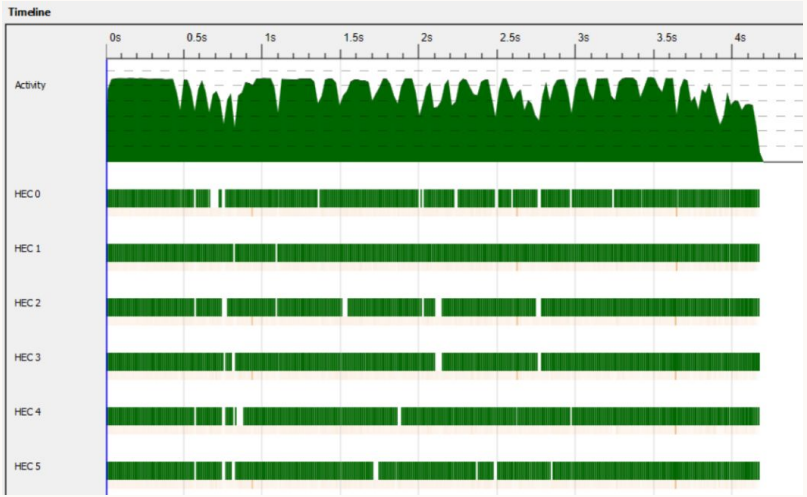
B

Spark Conversion Ratio vs. Threads



Threadscope Analysis

- When generating the Threadscope diagram from our different runs we found that the Activity between the different threads used for a given run was generally well *balanced*
- Generally from our experimentation we found that every thread was typically busy throughout an entire run, suggesting that our parallelization was *efficient* at dividing the computational work to allow for performance improvement



Conclusion

1. Speedup (avg: 3.851003774, max: 5.6986046975)
    - The parallel solver achieved an considerable speedup, with a reasonable depth and threads.
    - Speedup increased with search tree depth and thread, and achieved a peak.
2. Spark Conversion Ratio (avg: 0.2027896166, max: 0.4436936937)
    - The average spark conversion increase with the increase usage of depth and threads.
3. Threadscope Analysis
    - Balanced workloads among threads

Thank You!

Works Cited

- https://static.wikia.nocookie.net/newclubpenguin/images/f/ff/Mancala_Layout.png/revision/latest?cb=20200807100804
- https://museuartesacra.org.br/wp-content/uploads/2020/04/Tabuleiro-Mancala_Sentido-do-jogo_OK-1024x472.png
- https://www.coolmathgames.com/blog/how-to-play-mancala