Mancala Parallelization Using Minimax: Functional Programming Approach

Daniel Manjarrez (dam2274), Caiwu Chen (cc4786), Sindhu Krishnamurthy (sk4699)

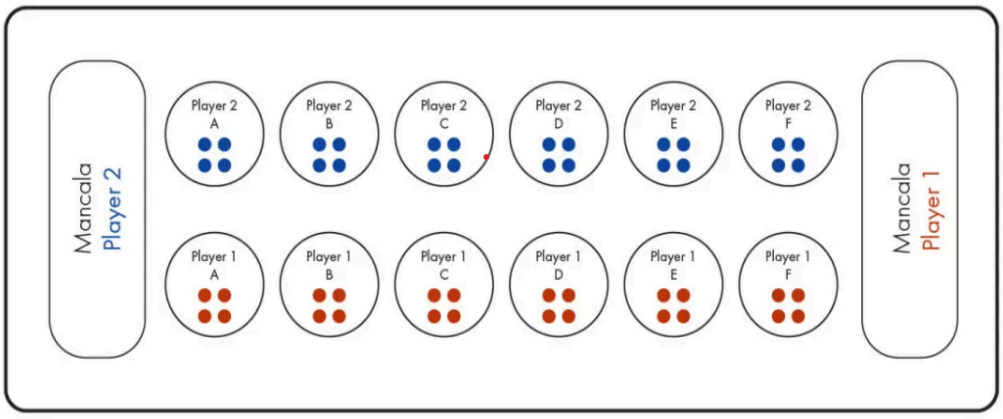
Dec 18, 2024

1. Introduction

This report describes a parallel Haskell implementation of a Mancala game solver using adversarial search via the Minimax decision-making algorithm aided by the Alpha–Beta Pruning search algorithm, an approach commonly used in two player turn-based zero-sum games.

Mancala is played on a board consisting of 12 small pits and 2 larger pits called “stores” or “mancalas.” Each player controls 6 small pits on their side and has one store to collect captured stones. During gameplay, players take turns picking up all the stones from one of their pits and “sowing” them into subsequent pits one by one counterclockwise. Players may sow stones into their own store, but skip the opponent's store. If the last stone lands in the player’s store, they get another turn; otherwise, control passes to the opponent. If the last stone lands in an empty pit on the player’s side, they capture all stones from the opponent’s directly opposite pit. The game ends when all pits on one player's side are empty; at this point, any remaining stones are collected into the opponent’s store. The player with the most stones in their store wins.

Section 2 includes an overview of the Adversarial Search Algorithms along with details of implementation for the Mancala game solver in Haskell. Section 3 introduces three versions of parallelism being applied to the sequential implementation, which together aimed to significantly improve the overall average performance of the Mancala game solver.



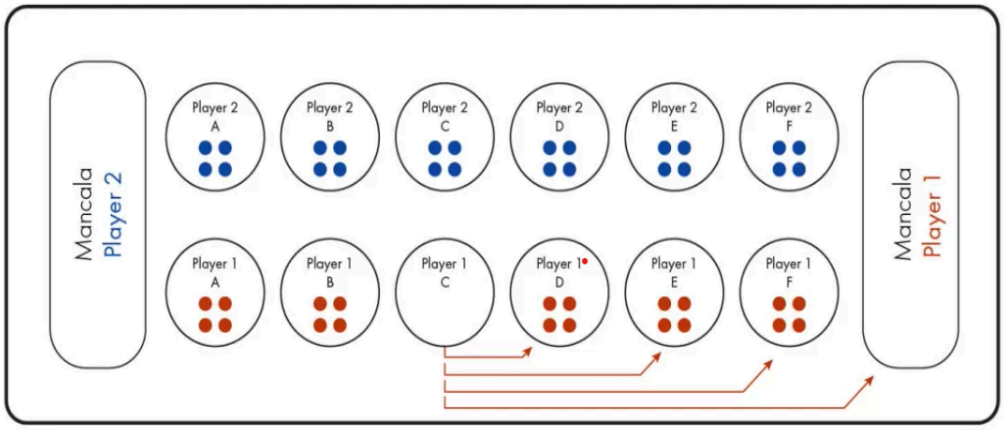


Figure 1. Mancala Board [1]

2. Adversarial Search Algorithms

2.1 *Minimax with Alpha-Beta Pruning*

We use the Minimax algorithm to make strategic decisions by evaluating potential moves and anticipating the opponent's responses. During gameplay, the algorithm recursively explores possible game states, alternating between the player and opponent, and the depth decrements with each recursive call. When the depth reaches zero, the game ends, or the player has no remaining moves, the evaluation of the board is returned based on a heuristic function. The scores from these leaf nodes of the tree are then propagated back up, with MAX nodes selecting the highest score and MIN nodes selecting the lowest. The optimal move is chosen based on the highest score at the root node. To reduce computation time during this process, we apply alpha-beta pruning to eliminate unnecessary branches from the search tree.

Alpha-beta pruning introduces two parameters: alpha, the maximum score that the maximizing player is guaranteed to achieve so far, and beta, the minimum score that the minimizing player is guaranteed to achieve so far. These parameters track the best possible scores for both the MAX and MIN players. For the maximizing player, alpha starts at -∞, and for the minimizing player, beta starts at ∞. During the search, alpha and beta are updated at each node based on the current best scores for the players. If the current node is a MAX node and its value exceeds beta, it means the minimizing player would not choose this move, so the branch is pruned. Similarly, if the current node is a MIN node and its value is less than alpha, then the

maximizing player would not choose this move, so the branch is pruned. Thus, the algorithm effectively avoids unnecessary computations.

2.2 *Haskell Implementation*

The program takes 2 arguments: `<depth>`, the depth of the search tree to be used for the minimax algorithm, and `<parallelDepth>`, the depth at which the minimax algorithm shifts from parallel to sequential. The program starts with the following command: `cabal exec Parallel-Minimax-Mancala <depth> <parallelDepth>`. For more detailed usage information, refer to Appendix A and the README file. The program outputs the board display after each player's move until the game ends and the winning player is displayed. By setting `<parallelDepth>` greater than `<depth>`, the program effectively runs sequentially (this approach is used for the parallel vs. sequential analysis discussed later); otherwise, it begins with a parallelized approach and switches to sequential once the `<parallelDepth>` threshold is reached. While the complete implementation of the Mancala game solver can be found in Appendix B, this section will focus on explaining the critical portions of the code.

The Mancala game solver implements all core mechanics of the game, including the logic for making moves, sowing and capturing seeds, and determining when the game has ended. However, the main driving logic for determining the best move lies in the `MiniMax.hs` file. The BestMove function calculates scores for each valid move and selects the move with the highest score for the maximizing player. It depends on the `minimax` function to help make this decision. In turn, the `minimax` function relies on `evaluateBoard` to calculate a heuristic score for a given board.

The `evaluateBoard` function:

```
Unset
evaluateBoard :: GameState -> Int
evaluateBoard (GameState b Player1) = b !! 6 - b !! 13
evaluateBoard (GameState b Player2) = b !! 13 - b !! 6
```

`evaluateBoard` utilizes a simple heuristic: From the point of view of Player1, how many more seeds are in Player1's store compared to Player2's store; and from the point of view of Player2, how many more seeds are in Player2's store compared to Player1's store. There remains plenty of room for improvement in this heuristic function; other valuable considerations include captures and extra turns. However, given the scope of the project and its specific focus on the impact of parallelization, a straightforward heuristic is used.

In order to help evaluate the best move for a player, the `minimax` function utilizes the principle variation method - specifically, the Young Brothers Wait Concept (YBWC). Based on this approach, the leftmost node, or "oldest brother," is evaluated sequentially with alpha-beta pruning; using these narrowed bounds, the remaining sibling nodes are evaluated in parallel [3]. Thus, we are able to effectively lower the overhead of parallelization by still enabling a degree of pruning to occur. While the `minimax` function encompasses the YBWC logic, whenever a subtree needs to be evaluated with sequential minimax, it calls a separate `seqMinimax` function.

The `minimax` function:

```
Unset
minimax :: GameState -> Int -> Bool -> Int -> Int -> Int -> Int
minimax state depth maximizingPlayer alpha beta parallelDepth
  | depth == 0 || isGameOver state || null validMovesList = evaluateBoard state
  | depth >= parallelDepth =
      let (firstMove:restMoves) = validMovesList
      firstValue = seqMinimax (makeMove state firstMove) (depth - 1) (not
maximizingPlayer) alpha beta
      (newAlpha, newBeta) = if maximizingPlayer
                            then (max alpha firstValue, beta)
                            else (alpha, min beta firstValue)
      values = parMap rdeepseq
                  (\pit -> minimax (makeMove state pit) (depth - 1) (not
maximizingPlayer) newAlpha newBeta parallelDepth)
```

```
                    restMoves
        combined = firstValue : values
        in if maximizingPlayer then maximum combined else minimum combined
  | otherwise =
        let values = map (\pit -> seqMinimax (makeMove state pit) (depth - 1)
(not maximizingPlayer) alpha beta) validMovesList
        in if maximizingPlayer then maximum values else minimum values
  where
        validMovesList = validMoves state
```

The function's base case ensures that if the depth reaches 0, the game is over, or there are no moves left, the board is evaluated to obtain a "score." When the depth is greater than or equal to the specified `<parallelDepth>`, however, we proceed with the YBWC approach. The first valid move is evaluated sequentially using `seqMinimax`, which generates an initial alpha or beta value depending on whether it is the maximizing or minimizing player's turn. After that, the sibling nodes - corresponding to the remaining valid moves - are recursively evaluated in parallel using `parMap` and `rdeepseq` to call `minimax`.

The results are then combined with the result of the first move, and the function returns either the maximum score for the maximizing player or the minimum score for the minimizing player. In this way, scores from the leaf nodes of the tree - generated by the base case - propagate up the tree through recursive calls, allowing each level of the tree to combine and compare results to determine the optimal move.

For depths below `<parallelDepth>`, `minimax` calls `seqMinimax` to process all moves sequentially.

The `seqMinimax` function:

```
Unset
-- seqMinimax
seqMinimax :: GameState -> Int -> Bool -> Int -> Int -> Int
seqMinimax state depth maximizingPlayer alpha beta
```

```
  | depth == 0 || isGameOver state || null validMovesList = evaluateBoard state
  | otherwise = alphaBeta validMovesList (alpha, beta)
  where
      validMovesList = validMoves state
      alphaBeta [] (a, b) = if maximizingPlayer then a else b
      alphaBeta (pit:pits) (a, b) =
      let newValue = seqMinimax (makeMove state pit) (depth - 1) (not
maximizingPlayer) a b
      (newAlpha, newBeta) = if maximizingPlayer
                            then (max a newValue, b)
                            else (a, min b newValue)
      in if newAlpha >= newBeta
      then if maximizingPlayer then newAlpha else newBeta
      else alphaBeta pits (newAlpha, newBeta)
```

`seqMinimax` shares the same base case as `minimax`, since it too must evaluate the board once the depth has reached 0, the game is over, or there are no more valid moves. Otherwise, the function evaluates each move by recursively calling itself on the game state resulting from that move, decrementing the depth and alternating the player. If the current player is the maximizingPlayer, alpha is updated if the new value is larger; otherwise, beta is updated if the new value is smaller. If alpha becomes greater than or equal to beta, the branch is pruned and no further moves are explored for the current state. Otherwise, the function continues by recursively exploring the remaining moves in the current branch and updating alpha and beta values as needed.

3. Parallelization

Our approach toward parallelization underwent several iterations. Before parallelizing, we implement an elementary sequential version of the algorithm in `MancalaSolver.hs` (not used in later comparison due to changes in implementation). Then, in the first iteration, we firstly parallelize computing the best move at each state without considering parallelized depth.

The full code is in `ParaMancala1.hs`. In the second version, we introduce parallelized depth using hybrid sequential and parallelization. At deeper levels, the algorithm computes the values for all possible moves in parallel. While at shallower levels, the algorithm falls back to the standard sequential alpha-beta pruning. The full code is in `ParaMancala2.hs`. In the third version, we reverse the hybrid execution order. At shallower levels, the algorithm computes parallelly; at deeper levels, the algorithm computes sequentially. The full code is in `ParaMancala3.hs`. In the last version, whose code lies in `src/`, we introduce the idea of principal variation search [3]. Each level has the leftmost branch evaluated sequentially to provide alpha beta cutoffs for the remaining branches to be evaluated in parallel.

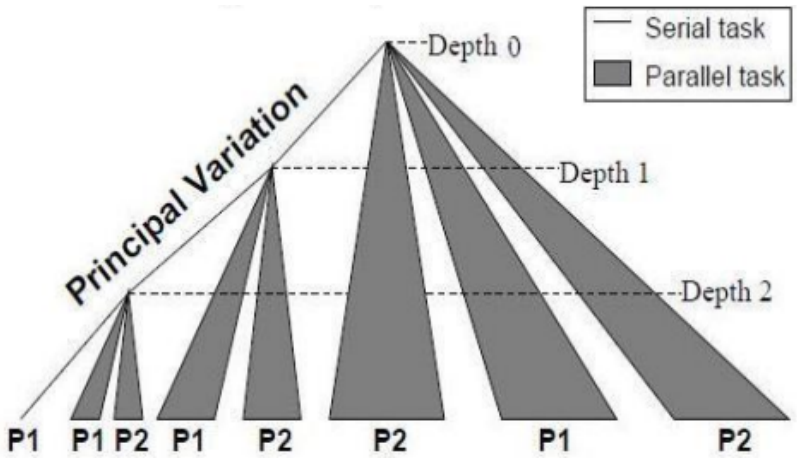


Figure 2. Principal Variation graph [3]

3.1 Parallel Evaluation of Rest of the Moves in `minimax`

Parallelizing the rest of the moves speeds up the computation when there are multiple valid moves to evaluate at deeper levels of the minimax tree. The full code can be found in `MiniMax.hs`.

```
Unset
...
values = parMap rdeepseq
                      (\pit -> minimax (makeMove state pit) (depth - 1) (not
maximizingPlayer) newAlpha newBeta parallelDepth)
                      restMoves
...
```

The `parMap` function applies the provided lambda function to each element in the list of valid moves, evaluating them in parallel. Each evaluation involves recursively calling the `minimax` function, and the game state is updated by making a move. Parameters like search depth, player type, and alpha-beta bounds are adjusted. `rdeepseq` ensures that each move's evaluation is fully computed before it is used, improving performance by leveraging parallelism.

3.2 Parallel Evaluation of All Moves in `bestMove`

Since the number of valid moves is typically small at the root level (e.g., 6-7 moves in games like Mancala), parallelizing this computation ensures fast decision-making without introducing significant overhead. The full code can be found in `MiniMax.hs`:

```
Unset
...
then parMap rdeepseq
                            (\pit -> (pit, minimax (makeMove state pit) (depth -
1) False (-1000) 1000 parallelDepth))
                            moves
...
```

For each move, it computes a tuple of the moves and its corresponding minimax value. The `minimax` function is applied to a new game state with the search depth reduced by 1, player switched, and alpha-beta bounds set to a reasonable range. The parameter `<parallelDepth>` controls the depth of parallel search, and `rdeepseq` ensures that each evaluation is fully evaluated before use, ensuring that all computations are completed before proceeding.

4. Performance Evaluation

We evaluated the performance of our parallelization efforts in our solver implementation and drew conclusions of its performance based on these aspects: Speedup as the depth of the search tree grew; spark conversion ratio as the depth of the search tree grew; speedup as the number of threads grew; spark conversion ratio as the number of threads grew; and the

Threadscope eventlog. Speedup naturally lends itself well as a comparison to the sequential version of the game solver both in the context of the parallel version's efficiency as the search tree grows and of the parallel version's efficiency as the number of threads used grows. Spark conversion ratio helps provide more context to the parallel version's performance potential as the search tree grows and as the number of threads used grows. Together, the speedup and spark conversion ratio help evaluate the parallel solver in comparison to the sequential one in terms of runtime, as well as evaluate its performance in the quality of its parallelization during a given runtime. According to our data, significant reduction in overall runtime was found with more room for optimization.

When it came to the speedup as the depth of the search tree grew, we experienced the speedup increase more and more as the depth of the search tree got larger, but then peak after a search tree of size 12, where it then started to decrease more and more.

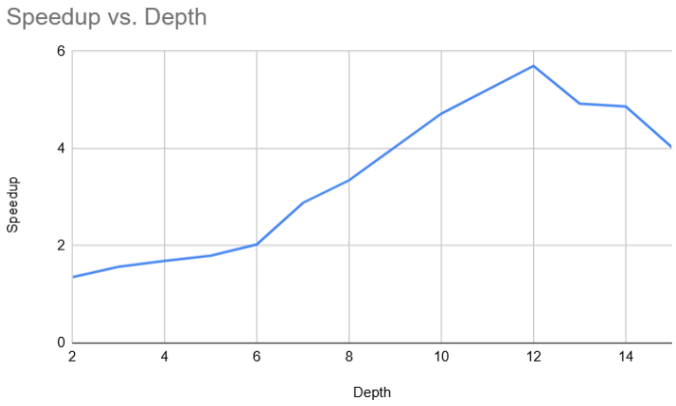


Figure 3. Speedup corresponding to depth

When it came to the spark conversion ratio as the depth of the search tree grew, we experienced the ratio increase more and more as the depth of the search tree got larger, but then start to plateau after a search tree of size 12, where it started to increase less and less.

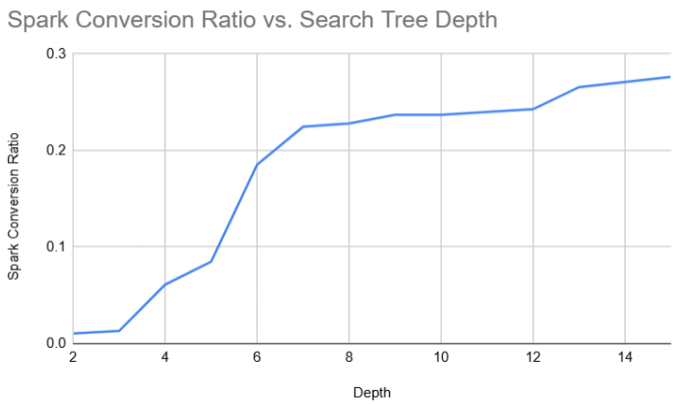


Figure 4. Spark Conversion Ratio corresponding to depth

When it came to the speedup as the number of threads used grew, we experienced the speedup increase more and more as we increased the number of threads used, but then start to plateau after more than 6 threads for smaller depth search trees and after 9 threads for larger depth search trees, where it started decreasing little by little.

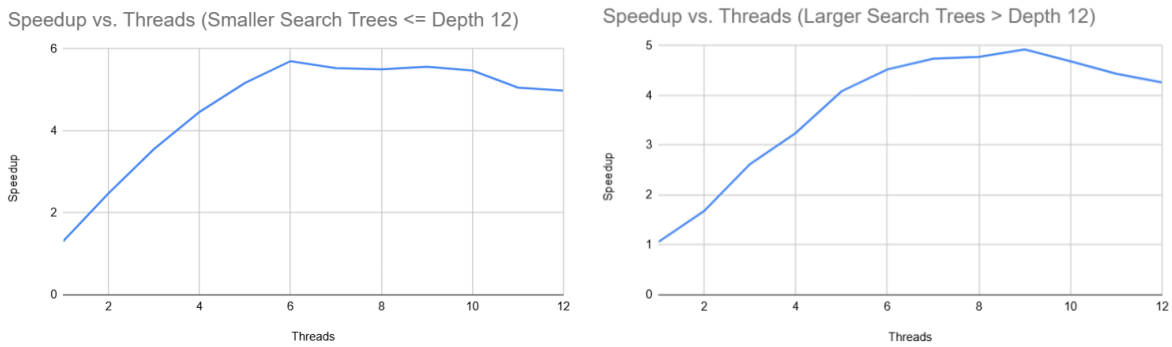


Figure 5. Speedup corresponding to threads and depth

When it came to the spark conversion ratio as the number of threads used grew, we experienced the ratio increase more and more as we increased the number of threads used, and this was the one plot where we did not observe a very evident decrease or plateau.

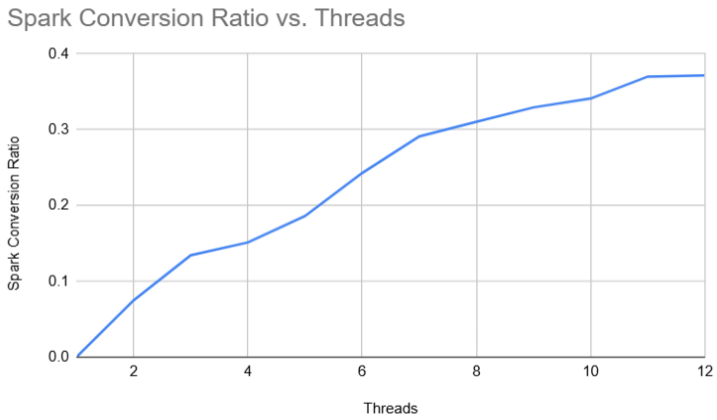


Figure 6. Spark Conversion Ratio corresponding to threads

The event log generated with Threadscope demonstrated that the parallel solver had a fairly equal and well-balanced activity workload divided among all the threads that were used at a given runtime with constant activity, as seen in the figure below:

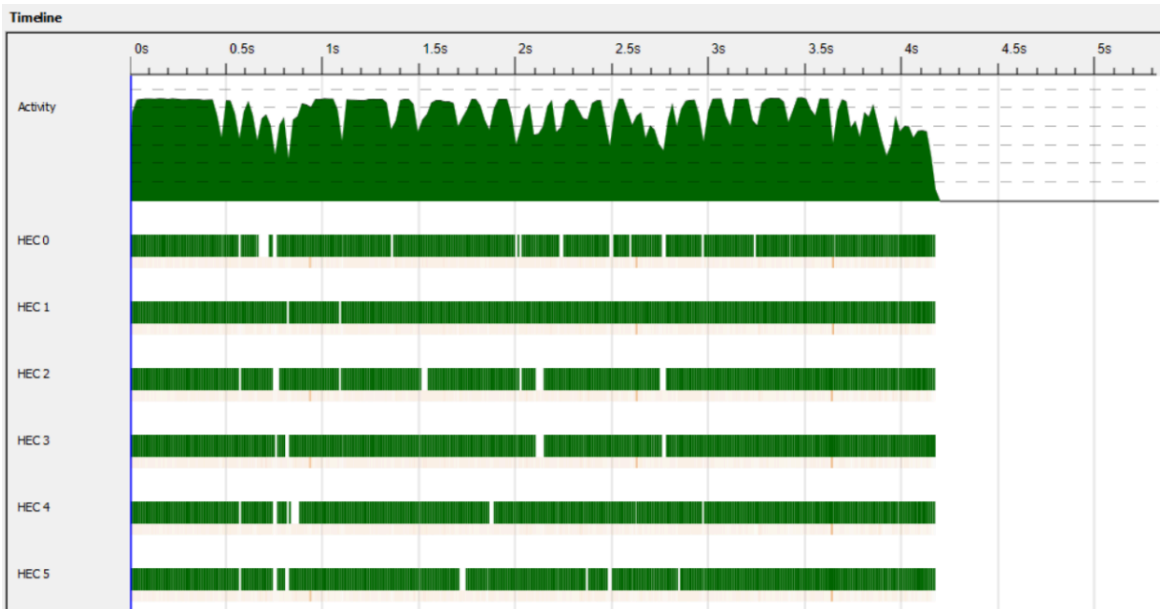


Figure 7. ThreadScope Analysis

Overall, we found that the best performance for the parallel solver in terms of speedup was when 6 threads were used on a search tree of 12 with `<parallelDepth>` being set to 10, with an overall speedup of 5.6986046975. In terms of the highest quality parallelization performed in terms of sparks converted, we observed that with all 12 of the threads at our

disposal used on a search tree of size 15 with `<parallelDepth>` set to 13, the overall spark conversion ratio was 0.4436936937. When it came to optimizing runtime for any given depth of search tree, the `<parallelDepth>` worked best being set to 2 less than the depth of the search tree, while using only 6 threads for smaller depth search trees or only 9 threads for larger depth search trees. For obtaining the highest quality parallelization, i.e. the highest spark conversion ratio, increasing the number of threads used as well as increasing search tree depth helped while keeping `<parallelDepth>` set to 2 less than the depth of the search tree. Overall speedup gained from the parallel solver was significant with the average speedup among all testing being 3.851003774, with much more room for improvement; the average spark conversion ratio was 0.2027896166, which indicated that about 20.28% of the generated sparks were executed in parallel, suggesting that parallelism was not fully exploited. However, we did see significant performance improvement, further suggesting that the parallel tasks executed had a large impact.

5. Conclusion

By testing multiple versions of parallelism with different strategies, the resulting parallel Mancala solver achieves significant speedup despite a lower spark conversion ratio. With the capabilities provided by Haskell, effective parallelism can be introduced and integrated into the minimax algorithm with just a few lines of carefully placed code. This approach results in substantial performance improvements while leaving room for further optimization.

References

[1] M. Blake, *"Understanding the Rules of Mancala: Capture the Win,"* LoveToKnow, Jul. 9, 2019. [Online]. Available: https://www.lovetoknow.com/life/lifestyle/mancala-rules. Accessed: Nov. 17, 2024.

[2] A. S. Biermann, "Parallel Implementation and Optimization of the Minimax Algorithm with Alpha-Beta Cutoffs in the Context of the Game Othello," *CRPC Summer Research Student, Caltech*, [Online]. Available: https://www.pressibus.org/ataxx/autre/minimax/node6.html. Accessed: Dec. 17, 2024.

[3] M. Guidry and C. McClendon, "Techniques to Parallelize Chess," [Online]. Available: https://ww2.cs.fsu.edu/~guidry/parallel_chess.pdf. Accessed: Nov. 17, 2024.

[4] "Young Brothers Wait Concept," Chessprogramming Wiki. [Online]. Available: https://www.chessprogramming.org/Young_Brothers_Wait_Concept. Accessed: Dec. 15, 2024.

Appendix A

[1] Parallel Solver's Usage:

Display.hs:

```
Unset
module Display where

import GameState (GameState(..))

-- Function to display the game board
displayBoard :: GameState -> IO ()
displayBoard (GameState b _) = do
  let player1Pits = take 6 b
      player1Store = b !! 6
      player2Pits = take 6 (drop 7 b)
      player2Store = b !! 13

  putStrLn "--------------------"
  putStrLn $ "| " ++ show player2Store ++ " | " ++ unwords (map show (reverse
player2Pits)) ++ " |"
  putStrLn "|------------------|"
  putStrLn $ "| " ++ unwords (map show player1Pits) ++ " | " ++ show
player1Store ++ " |"
  putStrLn "--------------------"
```

GameLogic.hs:

```
Unset
module GameLogic where

import GameState (GameState(..), Player(..), switchPlayer, Pit, Board)

sow :: GameState -> Int -> Int -> (Board, Int)
sow (GameState b p) start seeds = go b start seeds 0
  where
    go board _ 0 finalIdx = (board, finalIdx)
    go board idx n _ =
      let nextIdx = (idx + 1) `mod` 14
```

```
          shift = if p == Player1 && nextIdx == 13 || p == Player2 && nextIdx
== 6 then 1 else 0
          nextIdx' = (nextIdx + shift) `mod` 14
      in go (updateBoard board nextIdx' (+1)) nextIdx' (n - 1) nextIdx'
    updateBoard board idx f = take idx board ++ [f (board !! idx)] ++ drop (idx
+ 1) board

makeMove :: GameState -> Pit -> GameState
makeMove (GameState b p) pit =
  let seeds = b !! pit
      b1 = take pit b ++ [0] ++ drop (pit + 1) b  -- Remove seeds from the
selected pit
      (b2, finalIdx) = sow (GameState b1 p) pit seeds
      isOwnStore = (p == Player1 && finalIdx == 6) || (p == Player2 && finalIdx
== 13)
      isCapture = p == Player1 && finalIdx < 6 && b2 !! finalIdx == 1 && b2 !!
(12 - finalIdx) > 0
              || p == Player2 && finalIdx > 6 && finalIdx < 13 && b2 !!
finalIdx == 1 && b2 !! (12 - finalIdx) > 0
      b3 = if isCapture
           then captureSeeds b2 finalIdx p
           else b2
      nextPlayer = if isOwnStore then p else switchPlayer p
  in GameState b3 nextPlayer

captureSeeds :: Board -> Int -> Player -> Board
captureSeeds b idx player =
  let captured = b !! (12 - idx)
      b1 = take (12 - idx) b ++ [0] ++ drop (13 - idx) b
      b2 = take idx b1 ++ [0] ++ drop (idx + 1) b1
      storeIdx = if player == Player1 then 6 else 13
  in take storeIdx b2 ++ [(b2 !! storeIdx) + captured + 1] ++ drop (storeIdx +
1) b2

validMoves :: GameState -> [Pit]
validMoves (GameState b Player1) = [i | i <- [0..5], b !! i > 0]
validMoves (GameState b Player2) = [i | i <- [7..12], b !! i > 0]
```

GameState.hs:

```
Unset
module GameState where
```

```
type Pit = Int
type Board = [Int]
data Player = Player1 | Player2 deriving (Eq, Show)
data GameState = GameState { board :: Board, currentPlayer :: Player } deriving
(Show)

isGameOver :: GameState -> Bool
isGameOver (GameState b _) =
  all (== 0) (take 6 b) || all (== 0) (take 6 (drop 7 b))

switchPlayer :: Player -> Player
switchPlayer Player1 = Player2
switchPlayer Player2 = Player1
```

MiniMax.hs:

```
Unset
module MiniMax where

import GameLogic (makeMove, validMoves)
import GameState (GameState(..), Player(..), Pit, isGameOver)
import Control.Parallel.Strategies (parMap, rdeepseq)
import Data.List (maximumBy, uncons)
import Data.Function (on)


-- Heuristic to evaluate the board
evaluateBoard :: GameState -> Int
evaluateBoard (GameState b Player1) = b !! 6 - b !! 13
evaluateBoard (GameState b Player2) = b !! 13 - b !! 6

-- Minimax with alpha-beta pruning
minimax :: GameState -> Int -> Bool -> Int -> Int -> Int -> Int
minimax state depth maximizingPlayer alpha beta parallelDepth
  | depth == 0 || isGameOver state || null validMovesList = evaluateBoard state
  | depth >= parallelDepth =
    let (firstMove:restMoves) = validMovesList
        firstValue = seqMinimax (makeMove state firstMove) (depth - 1) (not
maximizingPlayer) alpha beta
        (newAlpha, newBeta) = if maximizingPlayer
                              then (max alpha firstValue, beta)
```

```
                              else (alpha, min beta firstValue)
        values = parMap rdeepseq
                 (\pit -> minimax (makeMove state pit) (depth - 1) (not
maximizingPlayer) newAlpha newBeta parallelDepth)
                 restMoves
        combined = firstValue : values
    in if maximizingPlayer then maximum combined else minimum combined
  | otherwise =
      let values = map (\pit -> seqMinimax (makeMove state pit) (depth - 1)
(not maximizingPlayer) alpha beta) validMovesList
      in if maximizingPlayer then maximum values else minimum values
  where
    validMovesList = validMoves state

seqMinimax :: GameState -> Int -> Bool -> Int -> Int -> Int
seqMinimax state depth maximizingPlayer alpha beta
  | depth == 0 || isGameOver state || null (validMovesList) = evaluateBoard
state
  | otherwise = alphaBeta (validMovesList) (alpha, beta)
  where
    validMovesList = validMoves state
    alphaBeta [] (a, b) = if maximizingPlayer then a else b
    alphaBeta (pit:pits) (a, b) =
      let newValue = seqMinimax (makeMove state pit) (depth - 1) (not
maximizingPlayer) a b
          (newAlpha, newBeta) = if maximizingPlayer
                                then (max a newValue, b)
                                else (a, min b newValue)
      in if newAlpha >= newBeta
         then if maximizingPlayer then newAlpha else newBeta
         else alphaBeta pits (newAlpha, newBeta)


bestMove :: GameState -> Int -> Int -> Pit
bestMove state depth parallelDepth =
  let moves = validMoves state
  in if null moves
     then error "No valid moves available"
     else let scores = if depth >= parallelDepth
                  then parMap rdeepseq
                           (\pit -> (pit, minimax (makeMove state pit) (depth -
1) False (-1000) 1000 parallelDepth))
                           moves
                  else map
```

```
                        (\pit -> (pit, minimax (makeMove state pit) (depth -
1) False (-1000) 1000 parallelDepth))
                        moves
     in fst $ maximumBy (compare `on` snd) scores
```

Run.hs:

```
Unset
module Run where

import GameState (GameState(..), isGameOver, Player(..), currentPlayer, board)
import Display (displayBoard)
import MiniMax (bestMove)
import GameLogic (makeMove, validMoves)

playGame :: GameState -> Int -> Int -> IO ()
playGame state depth parallelDepth
  | isGameOver state = do
      displayBoard state
      putStrLn "Game Over!"
      let finalBoard = board state
          -- Player 1's score is in pit 6
          player1Score = finalBoard !! 6 + sum (take 6 finalBoard)
          -- Player 2's score is in pit 13
          player2Score = finalBoard !! 13 + sum (take 6 (drop 7 finalBoard))
      putStrLn $ "Final Scores - Player 1: " ++ show player1Score ++ ", Player
2: " ++ show player2Score
      if player1Score > player2Score
        then putStrLn "Player 1 Wins!"
        else if player2Score > player1Score
          then putStrLn "Player 2 Wins!"
          else putStrLn "It's a tie!"
  | otherwise = do
      displayBoard state
      let move = bestMove state depth parallelDepth
      putStrLn $ "Player " ++ show (currentPlayer state) ++ " chooses pit " ++
show move
      let newState = makeMove state move
      playGame newState depth parallelDepth
```

Main.hs:

Unset

```
module Main where

import System.Environment (getArgs)
import Run (playGame)
import GameState (GameState(..), Player(..))

main :: IO ()
main = do
  args <- getArgs

  case args of
    [depthStr, parallelDepthStr] -> do
      let depth = read depthStr :: Int
          parallelDepth = read parallelDepthStr :: Int
      putStrLn "Starting Mancala Game"
      let initialState = GameState [4, 4, 4, 4, 4, 4, 0, 4, 4, 4, 4, 4, 4, 0]
Player1
      playGame initialState depth parallelDepth
    _ -> do
      putStrLn "Usage: ParaMancala3 <depth> <parallelDepth>"
```