Optimizing Halma: Parallel Minimax and Alpha-Beta Pruning in Haskell

Catherine Lyu (hl3553), Luana Liao (ll3637)

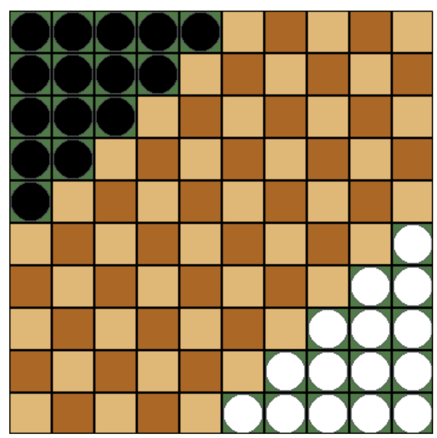
December 2024

1 Introduction

Halma is a strategic board game played on a checkerboard. Unlike the more well-known Chinese Checkers (Figure 1a), which can be played on a hexagonal board with up to six players, Halma is a two-player precursor invented in the 1800s, which uses square tiles. The goal of each player is to move all their pieces from their starting corner to the opponent's corner (Figure 1b).

Players take turns moving one piece per turn across the grid, either to an adjacent empty square or by jumping over an adjacent piece. Multiple consecutive jumps over other pieces are allowed in a single turn if the piece can land in an open space.

(a) A Chinese Checkers board.



(b) Example starting setup for 2-player Halma.

Figure 1: Game layout examples.

1.1 Movement Rules

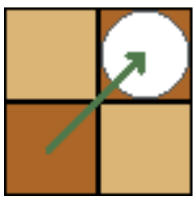
To elaborate, there are two ways to move in Halma:

- **Single Move**: A piece moves to an adjacent unoccupied square. This ends the player's turn.

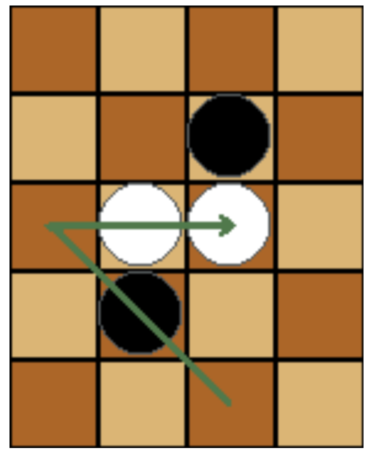
- **Jump**: A piece jumps over an adjacent piece (either the player's own or the opponent's) to land in a blank square on the other side. Consecutive jumps are allowed in a single turn, but they are optional. Players can stop jumping even if more jumps are available.

Pieces are never "captured" or removed from the board, even when jumped. Additionally, jumping is always optional and not enforced by the rules.

The rules and examples of movement mechanics were adapted from Pocket Monkey's Halma Guide [1]. This website also provided visual examples used in Figure 2.



(a) A piece making a single move.



(b) A piece making two jumps and stopping.

Figure 2: Examples of movement mechanics in Halma.

2 Project Overview

The Halma game presents an intriguing computational challenge due to its complex decision space and unique gameplay mechanics. The high branching factor of the game, especially with the possibility of multiple jumps in a single turn, makes it an ideal problem to study algorithms such as Minimax and optimizations such as parallelization. Beyond its gameplay, Halma offers an excellent opportunity to explore computational techniques and evaluate their performance in a real-world scenario.

In this project, our objective was to tackle the problem of efficiently solving Halma using Minimax, a classic algorithm for two-player games that evaluates potential moves by simulating decision making for both players. By combining Minimax with parallelization, we aimed to accelerate the search process, leveraging Haskell's functional programming paradigm and robust concurrency features. Our approach also incorporated alpha-beta pruning, a technique that reduces the search space by eliminating unproductive branches, further improving the efficiency of the algorithm.

The way an AI for Halma works is by representing the board as a game state and potential moves as transitions between states. The objective of such algorithms is to determine the optimal sequence of moves that would lead to victory, transferring all pieces from a starting camp to the opponent's camp.

The challenge lies in the exponential growth of the search space. The game's branching factor is high, especially due to its allowance for multiple jumps in a single turn.

If the algorithm is too slow or resource-intensive, it becomes impractical as a real-time adversary. By integrating parallelization, we aim to distribute computations across multiple cores, enhancing performance.

To address these challenges, we explored optimizations for the Minimax algorithm, a staple in two-player game AI. Specifically, our project goals included:

1. Implement a sequential version of Minimax to serve as a baseline.
2. Parallelization of the algorithm to distribute computational workload across multiple cores, reducing runtime.
3. Enhance the algorithm with alpha-beta pruning to cut off unproductive branches of the search tree.
4. Evaluate the effectiveness of these approaches by measuring runtime improvements and decision quality.

In this work, we designed a Halma-specific game state graph where nodes represent board configurations and edges represent possible moves. Using Haskell, we implemented these algorithms to test and optimize performance. Our solution leverages Haskell's powerful concurrency features, including the Par monad and Strategies library, to parallelize computations effectively.

We found that each optimization layer—from alpha-beta pruning to parallelized computation—yielded significant improvements in runtime while maintaining the quality of decisions. However, the project also revealed areas for further exploration, such as dynamic workload balancing and hybrid strategies.

In the following sections, we discuss the design and implementation of our algorithms, analyze their performance, and reflect on the challenges and future directions for Halma AI development.

2.1 Minimax Algorithm for Game Agents

The minimax algorithm is a fundamental approach for creating game-playing agents. It simulates all possible moves and counter-moves in a game, constructing a tree of game states. Each node in the tree represents a possible configuration of the game, and the edges represent the moves taken to reach those states. The goal is to identify the optimal move by evaluating the terminal states using a heuristic function.

2.1.1 Game State Representation

In our Halma AI implementation, we use a `GameState` class to store properties like the board configuration, current player, and move history. Here is a snippet from our code:

```
data Color = Black | White deriving (Eq, Show)

type Position = (Int, Int)

data Piece = Piece {
    position :: Position, -- (x,y) coord of piece
    color    :: Color -- which player's piece
```

```
} deriving (Eq, Show)

type Board = [[Maybe Piece]]

data GameState = GameState {
    board         :: Board
    currentPlayer :: Color
} deriving (Show)
```

To determine the best move, the Minimax algorithm explores all possible future game states, constructing a decision tree. At each level of the tree, it alternates between maximizing and minimizing the potential outcomes. The "minimax" value of a position represents the best achievable score if both players make the best possible moves.

The algorithm evaluates which moves lead to the best outcomes for each player using some sort of heuristic function.

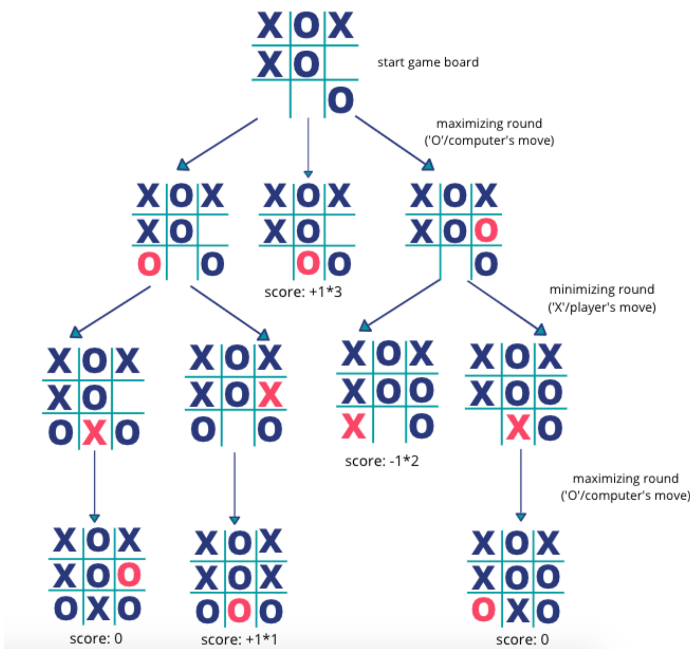


Figure 3: Minimax Example for TicTacToe

2.2 Heuristic Function

Since Halma has a large state space and may not be solved exhaustively within reasonable time limits, we rely on heuristic evaluations. One such heuristic is the Manhattan distance of pieces to their target zone. Specifically, we used the following evaluating function:

```
evaluateBoard :: Board -> Int evaluateBoard b = let whiteScore = sum [
    distance (position p) (0,0) | ... ] blackScore = sum [ distance (
  position p) (7,7) | ... ] in blackScore - whiteScore
```

This calculate the difference between the sum of black pieces' Manhattan distances from the bottom right corner and the sum of white pieces' Manhattan distances from the top left corner. The higher score means a bigger total distance for the black player or a smaller total distance for a white player, so a higher score favors white.

3 Sequential Solution

Starting board:

	0	1	2	3	4	5	6	7
0	B	B	B	B				
1	B	B	B					
2	B	B						
3	B							
4								W
5							W	W
6						W	W	W
7					W	W	W	W

Figure 4: Starting Board

	0	1	2	3	4	5	6	7
0	B			B				
1	B			B		B		
2	W	B	W					
3				B			W	
4			B			B		
5			W	W		B	W	
6							W	
7				W		W	W	

Figure 5: Mid Game Board

Initially, we implemented a basic minimax algorithm with alpha-beta pruning. The algorithm evaluates the board state and prunes branches that do not influence the final decision. This serves as the baseline for performance comparisons. The following describes in high level how the algorithm works:

- Enumerates all possible moves for the current player.
- Recursively applies Minimax to the resulting states.
- Uses alpha and beta bounds to prune unpromising branches.

4 Parallel Solutions

While alpha-beta pruning improved performance significantly, we wanted to further reduce the runtime by exploiting parallelism. Parallelization is natural here because evaluating different moves (child states) at a given level is often independent. Haskell's parallel constructs make it straightforward to spawn computations in parallel and combine results.

We explored two main parallelization strategies:

4.1 Top-Level Parallelism

In the *top-level parallelism* approach, we evaluate all immediate child states of the root node in parallel. The steps are:

1. At the root state, generate all possible moves for the current player.
2. Use parMap and rpar to run minimax on each child state concurrently.
3. Gather their evaluations and pick the best move.

This strategy can deliver substantial speedups if the branching factor at the root is large. However, top-level parallelism alone does not help once we move deeper into the tree, as later levels are evaluated sequentially (or less parallelized) once we commit to a branch. For our test configuration, we used a depth of 3 and ran on a machine with 12 or 8 cores.

Top-level parallelism is simple to implement and reason about. It reduces latency for one move decision if the root has many possible moves.

On the flip side, its benefits diminish if the number of top-level moves is small or if alpha-beta pruning quickly eliminates most moves, since this parallelism does not exploit deeper parallelism.

4.2 Chunked Parallelism

The *chunked parallelism* ideas are summarized as follows:

1. Split the top-level moves into chunks (e.g., groups of total number of possible moves / maximum core number each).
2. Evaluate one chunk in parallel, update global alpha-beta bounds, and potentially prune future chunks if the pruning condition is met.
3. Proceed to the next chunk only if necessary, using updated alpha and beta values.

This approach tries to combine parallelism with more effective pruning. By chunking, we do not waste resources evaluating all moves if a strong pruning opportunity arises early. We can stop processing remaining chunks once a prune condition (beta $\leq$ alpha) is reached, saving time.

A chunk size of number of all possible moves divided by 12 or 8 was used in our experiments depending on the maximum of cores of the machine. On a mid-state board with close to 500 possible moves at the top level, chunked parallelism allowed early pruning after evaluating the first few chunks, potentially speeding up the decision.

Chunked parallelism retains the benefit of parallelism while not committing to evaluating all moves upfront. It potentially reduces wasted computations by applying alpha-beta updates incrementally.

However, it is more complex to implement and requires careful tuning of chunk size. Too large a chunk behaves like the naive top-level parallelism and too small leads to overhead and less parallel efficiency.

4.3 Implementation Details

Both parallel approaches rely on similar code structures to the sequential version with parallel combinators used in the respective Minimax functions to make the computations parallel. The key difference is how the results are combined and how alpha-beta bounds are updated.

In top-level parallelism, we compute all moves at once and pick the best:

```
alphaBeta :: [GameState] -> (Int, GameState) -> Int -> Int -> Bool -> Int -> (Int, GameState)
alphaBeta [] bestEval _ _ _ _ = bestEval
alphaBeta gameStates (bestVal, bestState) alpha beta maximizingPlayer depth =
  let results = parMap rpar (\childState -> minimax childState (depth - 1) alpha beta (not maximizingPlayer)) gameStates
      evals = map fst results
      bestIndex = if maximizingPlayer
                  then snd $ maximumBy (\(v1,_) (v2,_) -> compare v1 v2) (zip evals [0..])
                  else snd $ minimumBy (\(v1,_) (v2,_) -> compare v1 v2) (zip evals [0..])
      bestEval = evals !! bestIndex
      chosenChild = gameStates !! bestIndex
      finalVal = if maximizingPlayer then max bestVal bestEval else min bestVal bestEval
      finalSt  = if (maximizingPlayer && bestEval > bestVal) || (not maximizingPlayer && bestEval < bestVal)
                 then chosenChild else bestState
  in (finalVal, finalSt)
```

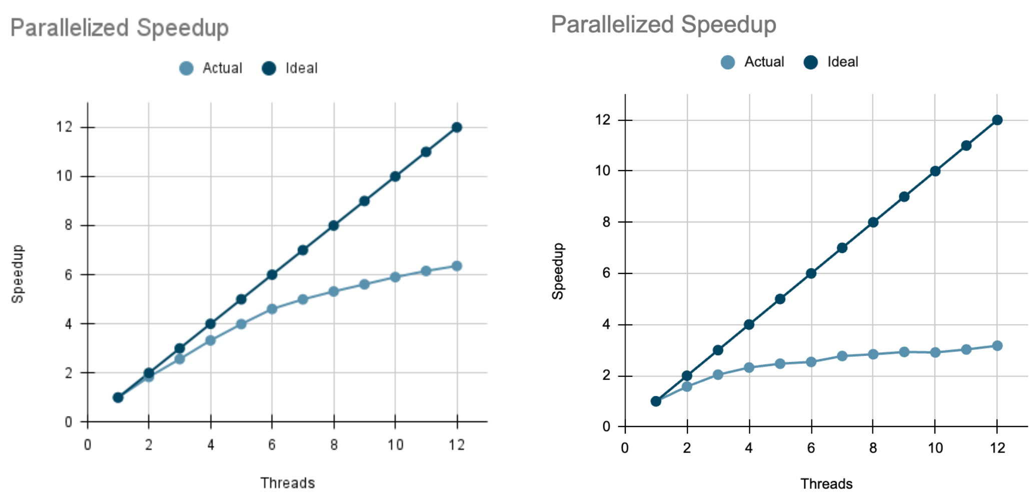
In chunked parallelism, we compute in batches, updating alpha and beta after each batch before deciding to continue:

```
alphaBetaChunked :: Int -> [GameState] -> (Int, GameState) -> Int -> Int -> Bool -> Int -> (Int, GameState)
alphaBetaChunked _ [] bestEval _ _ _ _ = bestEval
alphaBetaChunked cSize gs (bestVal, bestState) alpha beta maximizingPlayer depth =
  let chunks = chunkList cSize gs
  in processChunks chunks (bestVal, bestState) alpha beta
 where
  processChunks [] (curVal, curSt) _ _ = (curVal, curSt)
  processChunks (chunk:rest) (curVal, curSt) curA curB
      | curB <= curA = (curVal, curSt)
      | otherwise =
          let results = parMap rpar (\childState -> minimax childState (depth - 1) curA curB (not maximizingPlayer)) chunk
              evals = map fst results
              -- Pick best immediate childState from 'chunk', not from deeper states:
              bestIndex = if maximizingPlayer
                          then snd $ maximumBy (\(v1,_) (v2,_) -> compare v1 v2) (zip evals [0..])
                          else snd $ minimumBy (\(v1,_) (v2,_) -> compare v1 v2) (zip evals [0..])
              bestEval = evals !! bestIndex
              chosenChild = chunk !! bestIndex
```

```
                finalVal = if maximizingPlayer then max curVal
bestEval else min curVal bestEval
                finalSt  = if (maximizingPlayer && bestEval > curVal)
|| (not maximizingPlayer && bestEval < curVal)
                           then chosenChild else curSt
                newA = if maximizingPlayer then max curA finalVal else
 curA
                newB = if not maximizingPlayer then min curB finalVal
else curB
            in processChunks rest (finalVal, finalSt) newA newB
```

5 Performance Evaluation

We evaluated the two parallel solutions: top-level parallel, and chunked parallel across a variety of test states, and compared them to the sequential solution. We specifically focused on a generated mid-state test case. Our metrics include execution time for one move at a fixed depth (e.g., depth = 3) and speedup as we increase the number of cores.



(a) Top-level Parallelism Speedup (b) Chunked Parallelism Speedup

Figure 6: Speedup of (a) Top-level Parallelism and (b) Chunked Parallelism vs. Number of Cores

Top-Level Parallel Minimax												
Threads	1	2	3	4	5	6	7	8	9	10	11	12
Total Run Time(s)	2.473	1.345	0.964	0.744	0.620	0.537	0.495	0.465	0.441	0.419	0.402	0.389

Chunked Parallel Minimax												
Threads	1	2	3	4	5	6	7	8	9	10	11	12
Total Run Time(s)	0.457	0.290	0.224	0.197	0.185	0.180	0.165	0.161	0.156	0.157	0.151	0.144

Figure 7: Tables of Total Run Time of (a) Top-level Parallelism and (b) Chunked Parallelism vs. Number of Cores

As shown in Figure 6, while our parallel solutions do not achieve perfect linear scaling, the speedup continues to improve as we increase the number of cores, even beyond 12. This happens for both top-level and chunked parallelism, where top-level parallelism achieves a closer to ideal speedup curve. This suggests that, for a complex mid-state Halma board with a chaotic set of moves, the parallelization strategies can continue to leverage additional computational resources. This trend indicates that with further tuning or more substantial computational resources, the performance gains could become even better.

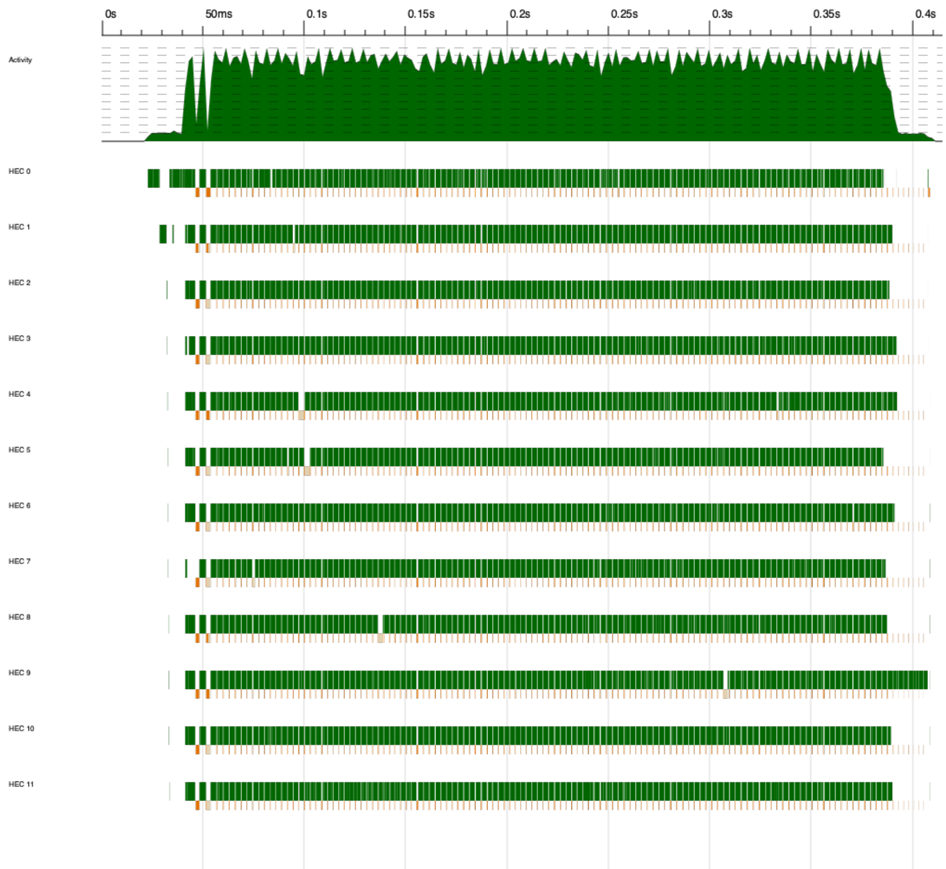


Figure 8: Threadscope of top-level parallelism on 12 cores

Figure 8 depicts a Threadscope visualization of top-level parallelism running on a 12-core machine. We see that all twelve cores remain busy for a significant portion of the runtime. Some cores start a bit earlier, likely due to initial task allocation, and one core eventually becomes responsible for aggregating and returning the results at the end. The relatively solid and continuous workload across all cores suggests that top-level parallelism effectively distributes initial moves at the root, allowing the system to exploit available parallel capacity. This distribution helps shorten the decision time, though some slight idle periods and synchronization points are visible as the computation nears completion.

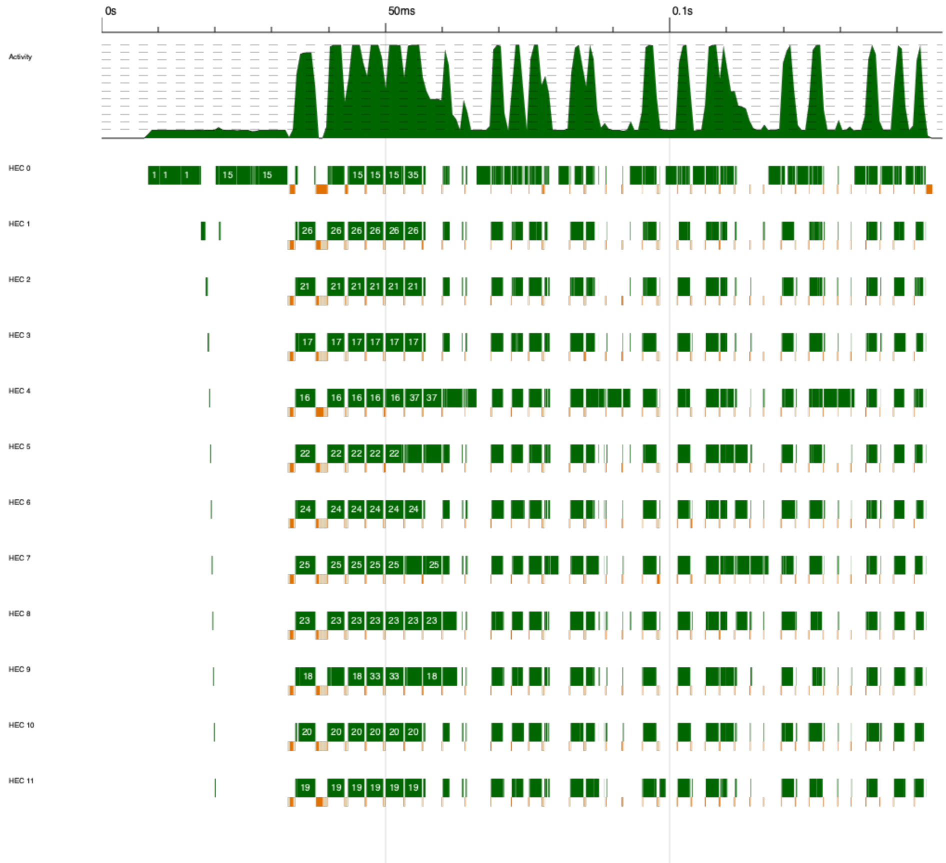


Figure 9: Threadscope of chunked parallelism on 12 cores

Figure 9 shows the Threadscope output for chunked parallelism using 12 cores. In this scenario, we observe more frequent transitions between active and idle phases on each core. This pattern aligns with the chunked parallel approach, where computations are done in batches. After each batch, alpha-beta global bounds are updated before proceeding to the next chunk. While the workload is still reasonably well spread out and cores remain active most of the time, these incremental synchronization steps lead to more intermittent workload patterns. Despite the pauses, the cores still do a balanced amount of work, and pruning between chunks can significantly cut down on unnecessary computations. Thus, chunked parallelism has shown to ubstantial speed improvements, especially if pruning occurs effectively.

	Avg. Run Time (s)
Sequential Minimax	14.81
Sequential Minimax + Alpha Beta Pruning	0.79
Top Level Parallelism (12 threads) + Latter Layers Alpha Beta Pruning	0.39
Chunk Parallelism with Global Bounds Updating Alpha Beta Pruning	0.15

Figure 10: Time comparison between the different methods

In Figure 10, we see a direct runtime comparison of the different methods. The sequential Minimax approach without alpha-beta pruning is very slow, and adding alpha-beta pruning drastically reduces execution time. Introducing parallelism at the top level further cuts the runtime, and chunked parallelism with global alpha-beta updating yields even better performance. The table highlights how each layer of optimization, including alpha-beta pruning, parallelism, and chunking, contributes to speed improvements. The final chunked approach runs substantially faster than the original sequential baseline, demonstrating the effectiveness of combining pruning with smartly managed parallel workloads.

6 Discussion and Further Considerations

6.1 Alpha-Beta Pruning

Alpha-beta pruning remains critical in reducing the search space. With parallelizing the top level of the search space, pruning the following layers remain important because it saves time by skipping obviously inferior moves. Therefore, with pruning, parallelism is more efficient since we evaluate fewer states overall.

6.2 Parallelism Trade-offs

While parallelization clearly improves performance, it brings the challenge to chunked parallelism because the algorithm requires iterative synchronization to update alpha-beta bounds.

7 Conclusion

We explored Minimax and alpha-beta pruning for the game Halma, and evaluated both sequential and parallel solutions. Parallel strategies, particularly top-level parallelism and chunked parallelism, showed promising runtime improvements over a pure sequential approach. While chunked parallelism adds complexity, it can yield better pruning effectiveness, reducing wasted work.

Overall, our results suggest that parallelization is a valuable tool for complex board games like Halma, but fitting alpha-beta pruning to the parallelized structures is also very important in reducing the size of the overall search space.

Appendix

Code Listing

Because our codes are lengthy with the game settings, we only included the code from halma_gameplay_ai_vs_input.hs, and the rest of the code can be found in our submission .tar.gz file.

```
import System.IO

-- Data Types and Constants
data Color = Black | White deriving (Eq, Show)

type Position = (Int, Int)

data Piece = Piece {
    position :: Position,
    color    :: Color
} deriving (Eq, Show)

type Board = [[Maybe Piece]]

data GameState = GameState {
    board         :: Board,
    currentPlayer :: Color
} deriving (Show)

rows, cols :: Int
rows = 8
cols = 8

blackStart, whiteStart :: [Position]
blackStart = [(0, 0), (0, 1), (0, 2), (0, 3),
              (1, 0), (1, 1), (1, 2),
              (2, 0), (2, 1),
              (3, 0)]

whiteStart = [(4, 7),
              (5, 6), (5, 7),
              (6, 5), (6, 6), (6, 7),
              (7, 4), (7, 5), (7, 6), (7, 7)]

-- Initialize Board
initializeBoard :: Board
initializeBoard = [ [ initialPieceAt (r, c) | c <- [0..cols-1] ] | r
    <- [0..rows-1] ]

initialPieceAt :: Position -> Maybe Piece
initialPieceAt pos
    | pos `elem` blackStart = Just (Piece pos Black)
```

```haskell
    | pos `elem` whiteStart = Just (Piece pos White)
    | otherwise             = Nothing

-- Access Board Elements
getPiece :: Board -> Position -> Maybe Piece
getPiece b (r, c) =
    if inBounds (r, c) then (b !! r) !! c else Nothing

inBounds :: Position -> Bool
inBounds (r, c) = r >= 0 && r < rows && c >= 0 && c < cols

-- Move Pieces
movePiece :: Board -> Position -> Position -> Board
movePiece b from to =
    let piece = getPiece b from
        updatedPiece = fmap (\p -> p { position = to }) piece
        b1 = updateBoard b from Nothing
        b2 = updateBoard b1 to updatedPiece
    in b2

updateBoard :: Board -> Position -> Maybe Piece -> Board
updateBoard b (r, c) val =
    take r b ++
    [take c (b !! r) ++ [val] ++ drop (c + 1) (b !! r)] ++
    drop (r + 1) b

-- Generate Valid Moves
directions :: [Position]
directions = [(dr, dc) | dr <- [-1,0,1], dc <- [-1,0,1], (dr, dc) /= (0, 0)]

getValidMoves :: Board -> Piece -> [Position]
getValidMoves b p =
    let singleMoves = getSingleMoves b p
        jumpMoves = getJumpMoves b (position p) []
    in singleMoves ++ jumpMoves

getSingleMoves :: Board -> Piece -> [Position]
getSingleMoves b p =
    [ (r, c)
    | (dr, dc) <- directions
    , let (r, c) = addPos (position p) (dr, dc)
    , inBounds (r, c)
    , isEmpty b (r, c)
    ]

addPos :: Position -> Position -> Position
addPos (r1, c1) (r2, c2) = (r1 + r2, c1 + c2)

isEmpty :: Board -> Position -> Bool
isEmpty b pos = getPiece b pos == Nothing

getJumpMoves :: Board -> Position -> [Position] -> [Position]
getJumpMoves b pos visited =
    concatMap (\dir -> jumpInDirection b pos dir (pos : visited)) directions
```

```haskell

jumpInDirection :: Board -> Position -> Position -> [Position] -> [
    Position]
jumpInDirection b (r, c) (dr, dc) visited =
    let midPos = (r + dr, c + dc)
        landingPos = (r + 2*dr, c + 2*dc)
    in if inBounds landingPos &&
          not (landingPos `elem` visited) &&
          not (isEmpty b midPos) &&
          isEmpty b landingPos
       then
          let newVisited = landingPos : visited
              furtherJumps = getJumpMoves b landingPos newVisited
          in landingPos : furtherJumps
       else []

-- Check for Game Over
isGameOver :: Board -> Maybe Color
isGameOver b
    | allPiecesInZone b White blackStart = Just White
    | allPiecesInZone b Black whiteStart = Just Black
    | otherwise                          = Nothing

allPiecesInZone :: Board -> Color -> [Position] -> Bool
allPiecesInZone b colorPiece zone =
    let pieces = [ p | row <- b, Just p <- row, color p == colorPiece
    ]
    in not (null pieces) && all (\p -> position p `elem` zone) pieces

-- Evaluate Board
evaluateBoard :: Board -> Int
evaluateBoard b =
    let whiteScore = sum [ manhattanDistance (position p) (0, 0) | row
    <- b, Just p <- row, color p == White ]
        blackScore = sum [ manhattanDistance (position p) (7, 7) | row
    <- b, Just p <- row, color p == Black ]
    in blackScore - whiteScore -- Lower score favors White

manhattanDistance :: Position -> Position -> Int
manhattanDistance (r1, c1) (r2, c2) = abs (r1 - r2) + abs (c1 - c2)

-- Minimax Algorithm with Alpha-Beta Pruning
minimax :: GameState -> Int -> Int -> Int -> Bool -> (Int, GameState)
minimax gameState depth alpha beta maximizingPlayer =
    let b = board gameState
    in case isGameOver b of
        Just winner -> if winner == White then (10000, gameState) else
    (-10000, gameState)
        Nothing ->
            if depth == 0
            then (evaluateBoard b, gameState)
            else
                let moves = getAllMoves gameState
                    initialEval = if maximizingPlayer then (minBound,
    gameState) else (maxBound, gameState)
```

```haskell
                in alphaBeta moves initialEval alpha beta maximizingPlayer depth

alphaBeta :: [GameState] -> (Int, GameState) -> Int -> Int -> Bool -> Int -> (Int, GameState)
alphaBeta [] bestEval _ _ _ _ = bestEval
alphaBeta (gameState:rest) (bestVal, bestState) alpha beta maximizingPlayer depth =
    let (eval, _) = minimax gameState (depth - 1) alpha beta (not maximizingPlayer)
        (newBestVal, newBestState) =
            if maximizingPlayer
            then if eval > bestVal then (eval, gameState) else (bestVal, bestState)
            else if eval < bestVal then (eval, gameState) else (bestVal, bestState)
        newAlpha = if maximizingPlayer then max alpha eval else alpha
        newBeta  = if not maximizingPlayer then min beta eval else beta
    in if newBeta <= newAlpha
       then (newBestVal, newBestState)  -- Prune remaining moves
       else alphaBeta rest (newBestVal, newBestState) newAlpha newBeta maximizingPlayer depth

getAllMoves :: GameState -> [GameState]
getAllMoves gameState =
    let b = board gameState
        colorPiece = currentPlayer gameState
        pieces = [ p | row <- b, Just p <- row, color p == colorPiece ]
        moves = [ (p, dest) | p <- pieces, dest <- getValidMoves b p ]
        nextPlayer = switchPlayer colorPiece
        gameStates = [ GameState (movePiece b (position p) dest) nextPlayer | (p, dest) <- moves ]
    in gameStates

switchPlayer :: Color -> Color
switchPlayer Black = White
switchPlayer White = Black

-- Command-Line Interface
displayBoard :: Board -> IO ()
displayBoard b = do
    putStrLn "  0 1 2 3 4 5 6 7"
    mapM_ displayRow (zip [0..] b)

displayRow :: (Int, [Maybe Piece]) -> IO ()
displayRow (i, row) = do
    putStr (show i ++ " ")
    putStrLn $ concatMap displayCell row

displayCell :: Maybe Piece -> String
displayCell Nothing                 = ". "
displayCell (Just (Piece _ Black))  = "B "
displayCell (Just (Piece _ White))  = "W "

```

```haskell
-- Main Game Loop
main :: IO ()
main = do
    hSetBuffering stdout NoBuffering
    let initialState = GameState initializeBoard Black
    gameLoop initialState

gameLoop :: GameState -> IO ()
gameLoop gameState = do
    displayBoard (board gameState)
    case isGameOver (board gameState) of
        Just winner -> putStrLn $ show winner ++ " wins!"
        Nothing     -> do
            if currentPlayer gameState == Black
            then playerTurn gameState
            else aiTurn gameState

playerTurn :: GameState -> IO ()
playerTurn gameState = do
    putStrLn "Your turn. Enter move as 'row col newRow newCol':"
    input <- getLine
    let inputs = words input
    if length inputs == 4
    then case map read inputs :: [Int] of
        [row, col, newRow, newCol] ->
            let piece = getPiece (board gameState) (row, col)
            in case piece of
                Just p -> do
                    let validMoves = getValidMoves (board gameState) p
                    if (newRow, newCol) `elem` validMoves
                    then do
                        let newBoard = movePiece (board gameState) (row, col) (newRow, newCol)
                            newGameState = GameState newBoard White
                        gameLoop newGameState
                    else do
                        putStrLn "Invalid move. Try again."
                        playerTurn gameState
                Nothing -> do
                    putStrLn "No piece at that position. Try again."
                    playerTurn gameState
        _ -> do
            putStrLn "Invalid input format. Try again."
            playerTurn gameState
    else do
        putStrLn "Invalid input format. Try again."
        playerTurn gameState

aiTurn :: GameState -> IO ()
aiTurn gameState = do
    putStrLn "AI is thinking..."
    let (eval, newGameState) = minimax gameState 2 (minBound :: Int) (maxBound :: Int) True
    putStrLn $ "AI evaluated the board with a score of: " ++ show eval
    gameLoop newGameState
```

References

[1] Pocket Monkey. *Help: Halma*. Available at: http://www.pocket-monkey.com/help-halma.jsp

[2] Nuevo Foundation. *TicTacToe Minimax*. https://workshops.nuevofoundation.org/java-tictactoe/activity-5/

[3] Cameron Chafin. *HalmaGame*. GitHub Repository. Available at: https://github.com/cameronchafin/HalmaGame