AlphaGambit: Parallelizing MiniMax for Chess

Anthony Zhou

December 21, 2023

1 INTRODUCTION

1.1 Our Work

In the past, chess was always inherently a two-player game. But what if you wanted to practice alone? Unless you were simply studying theory, you had no way to play a game without a human partner. This all changed recently, with the advent of computer-based chess engines, which can intelligently pick moves to play a human. Beyond enabling the individual practice use case, these algorithms also give us insights into designing artificial intelligence in general.

The way that these chess engines traditionally work is by representing the chess board as a game state, and representing moves as transitions. The goal of these algorithms is to find an optimal path (i.e., an optimal move) through future game states to secure victory. However, this approach comes with a big problem, which is computational complexity. The search space scales exponentially with the number of moves you look ahead to. If the algorithm becomes either too slow or memory-intensive, then it is no longer useful as a real-time adversary. Thus, practical algorithms for playing chess (the canonical one being Minimax) typically only look a few steps ahead of the current state.

In this work, we are addressing this explosion of the search space by introducing several optimizations for the Minimax algorithm in Haskell. In particular, our goals are:

1. Implement a sequential version of the Minimax algorithm
2. Implement a parallelized version of the Minimax algorithm
3. Implement alpha-beta pruning, a common technique for reducing the size of the search space
4. Implement Jamboree, a hybrid technique that combines ideas from alpha-beta pruning with parallelized minimax for even faster running time.

To test and use these algorithms, we also created a state graph (based on the rules of chess) with nodes corresponding to game states and edges corresponding to moves. Finally, we implemented a visualization function that represents board states using Unicode chess characters.

We found that each of our optimizations achieved runtime improvements over Minimax, but also discovered some issues that, if solved, would unlock much greater performance increases. In the following report, we will share:

1. How we designed and implemented our algorithm
2. Runtime and profiling results and analysis
3. Discussion of challenges and opportunities
4. A full listing of the source code.

1.2 History of Chess

Chess first emerged in India during the 6th century AD as a game called ”Chaturanga.” This early version represented the various elements of a military force, including infantry, cavalry, elephants, and chariots. The game eventually spread through Persia, the Arab world, and Europe, where it evolved into its recognizable modern form in Southern Europe during the 15th century (7).

Today, chess has transcended borders and cultures to become a renowned game of strategy and intellectual prowess. The prominence of professional tournaments, notably the prestigious World Chess Championship, serves as a testament to chess's enduring global appeal. Chess remains an ever-evolving pursuit: technological innovations like artificial intelligence continually revolutionize the game, influencing strategies and gameplay (5).

1.3 Chess in Computing

Alan Turing first introduced the integration of chess with computing in the 1950s. Early computer programs of the 1960s built on Turing's proposals, executing rudimentary chess maneuvers.

The evolution of artificial intelligence (AI) in chess facilitated pivotal developments. Sophisticated algorithms, notably alpha-beta pruning, surfaced during the 1970s and 1980s, elevating the computational understanding of chess strategies. In the 1990s, IBM's Deep Thought and Deep Blue represented pivotal advancements in AI-driven chess computation (bri).

The 2000s ushered in a new era marked by the integration of machine learning methodologies into chess programs. Notably, the infusion of deep learning techniques, exemplified by neural networks like AlphaZero, revolutionized chess strategy and gameplay.

Renowned chess engines such as Stockfish, Komodo, and Houdini emerged, leveraging AI techniques to navigate the intricate complexities of chess. Monumental competitions, such as the historic Kasparov versus Deep Blue match, marked the convergence of computerized prowess against human intellect(8).

The integration of AI in chess profoundly impacted human engagement with the game. Chess programs became indispensable learning aids for players across skill levels. Furthermore, these programs facilitated in-depth analysis of game plays, enabling advanced scrutiny of moves and strategic maneuvers.

The pursuit of complete AI mastery in chess faces formidable challenges owing to the game's inherent complexity. Current exploration focuses on harnessing hybrid AI models to enhance comprehension and navigation of the intricate chess landscape.

1.4 Motivation

1.4.1 Overview of the Minimax Algorithm

Operating on the premise of sequential decision-making, Minimax seeks to optimize a player's move while considering the potential responses of an adversary: maximizing gains and minimizing losses.

Minimax utilizes a tree-like structure to analyze potential moves and their consequences. Each node in the tree represents a game state, branching out to potential future states based on possible moves. Minimax assigns players the roles of maximizing and minimizing adversaries. The maximizing player aims to maximize their advantage, while the minimizing player aims to minimize this advantage. The algorithm recursively evaluates each node in the tree, calculating a heuristic value that represents the potential outcome of the game from that state. Minimax backtracks up the tree after evaluating nodes and selecting the move that leads to the most favorable outcome based on the heuristics.

1.4.2 Challenges in Complexity
The primary challenge stems from the exponential growth in the number of nodes within the game tree as the depth increases. This exponential expansion leads to an astronomical number of potential game states, exponentially multiplying the computational resources required to explore all possibilities.

The resource-intensive sequential Minimax algorithm encounters limitations in exploring extensive game trees: as the tree's depth increases, the algorithm's capacity to conduct deep searches becomes limited, impacting its ability to navigate exponentially growing decision spaces. Traditional Minimax implementations are infeasible to exhaustively search the vast number of possible moves and resultant game states.

1.4.3 Alpha-Beta Pruning
Alpha-beta pruning, a pivotal optimization technique within Minimax, identifies and prunes trivial branches of the game tree. By discarding branches that do not impact the final decision, alpha-beta pruning significantly reduces the number of nodes to be evaluated without compromising the quality of the selected move, allowing Minimax to explore deeper into the game tree. For this section only, we used an implementation adapted from (6).

1.4.4 Advantages of Parallelism
By distributing tasks across multiple processors, parallel computing significantly enhances computation speed. Thus, parallelism provides a compelling solution to the limitations posed by sequential Minimax. Parallelism harnesses the power of multi-core architectures to mitigate the time-intensive nature of Minimax's exhaustive search.

1.4.5 Parallelism Leveraged through Haskell
Haskell's declarative and concurrent programming capabilities provide intrinsic support for parallelism (4), aligning seamlessly with the requirements of Minimax optimization. Haskell libraries such as Control.Parallel and Control.Concurrent includes essential tools and functionalities that empower developers to effectively implement parallelism within the Minimax algorithm.

1.4.6 Jamboree Algorithm
The Jamboree algorithm stands as a promising approach, particularly in the context of parallelization within Minimax optimization. Developed as an enhancement to Minimax, Jamboree aims to augment the algorithm's efficiency by leveraging parallel processing capabilities. Jamboree's structure inherently embodies a high degree of task independence, enabling concurrent execution of distinct branches of the game tree. This independence allows parallelization strategies to be efficiently implemented without substantial dependencies among individual tasks. We implement Jamboree following the pseudo-code given from chessprogramming's page.(2)

Unlike traditional Minimax where branch termination directly affects the overall search process, Jamboree's design allows for independent termination of branches. This characteristic facilitates more efficient parallelization by reducing the necessity for synchronization points, enabling better exploitation of parallel resources. The algorithm's design encourages simultaneous evaluation of multiple branches, harnessing parallel computing resources effectively. This simultaneous evaluation of branches aids in reducing computational time, enabling deeper exploration of the game tree within comparable time frames.

2 METHODS

To properly test our algorithms, we had to create some scaffolding around them. First, the search algorithms require some graph to search over, which we set up using functions from the `Chessica` library. They also require a heuristic function to judge how good intermediate game states are (since they can't reach the end of the game). Finally, we implemented a command-line interface for testing, profiling, and interacting with the various algorithms.

2.1 Structure

2.1.1 Chessica Package Integration

The implementation incorporates the Haskell package `Chessica`, designed to facilitate the creation and manipulation of chess board and game data types (3). It contains useful primitives for constructing a board, moving pieces, and checking for conditions like check and checkmate.

2.1.2 Heuristic Function

Because search functions – due to computational constraints – cannot search until the end state of a graph, they need to know if intermediate moves bring the player closer to a winning state. The heuristic function plays a critical role in the evaluation process, assigning a numerical score to individual board states. It achieves this by considering piece values and the control exerted by pieces over the board.

2.1.3 Search Functions

Employing the heuristic function alongside an initial board state, the methodology involves the construction and exploration of a tree representing potential moves and their subsequent board states. These search functions navigate through the tree structure, evaluating and selecting the most optimal move based on the heuristic score.

2.2 Parallelization

We used Control.Parallel.Strategies to parallelize the mapping of functions onto lists. When map is called, it allows the user to determine an evaluation method. The `minimaxPar` function harnesses parallelism within the Minimax algorithm, crucial for optimizing game decision-making. Central to its implementation is the use of `map`, which, when applied to a list of game states within `max_` and `min_` functions, generates a sequence of thunks. These thunks encapsulate potential game states and remain unevaluated until explicitly needed for further computation.

The key to parallelism lies in the evaluation strategy employed by `parList`, coupled with the `rseq` strategy. `parList` operates on a list of thunks, initiating parallel evaluation of computations within the list. Simultaneously, `rseq` orchestrates the reduction of thunks' results, ensuring they are computed in a specified sequence and enforcing thunks to a state known as weak normal form (WHNF).

Weak normal form denotes partial evaluation, where thunks are evaluated to their outermost constructor or lambda abstraction, without full computation. This partial evaluation readiness is crucial for efficient parallelism, as it sets the stage for controlled and predictable thunk evaluation. By using `rseq` to force thunks into WHNF, the algorithm orchestrates a sequence for thunk evaluation, effectively managing parallel execution.

2.2.1 Command Line Interface

The primary interface, defined in `app/Main.hs` and compiled to an executable, governs the input-output processing and serves as an intermediary to interact with the developed system. It facilitates the selection of search functions and specifies the desired depth for the search. Additionally, 'Main' orchestrates simulated chess games between distinct engines, aiding in the assessment and comparison of different strategies.

2.2.2 Visual Representations

Visual representation of the chessboard is achieved through the derivation of the `Show` typeclass for the Chessica board datatype, using Unicode characters for the chess pieces. Additionally, conversion from Unicode to image-based representations is accomplished using the Python Imaging Library (PIL), offering a visual rendition of the chessboard.

2.2.3 Testing and Performance Evaluation

To measure performance metrics and efficiency, the methodology incorporates Python and Bash scripting. These scripts conduct timed evaluations and collect performance data to gauge the efficacy of the implemented system.

2.3 Implementation

2.3.1 Chessica

The `generateGame` function creates a new game instance adhering to the standard chess rules. It initializes the game board and sets up the initial positions of the pieces.

The `allUpdates` function retrieves all potential moves available in a given game state, utilizing a specific rulebook. It evaluates and generates a list of all possible moves based on the defined rules.

The `nextStates` function, operating on a current game state, identifies and produces all feasible updates within the game. It generates a collection of potential subsequent game states following the standard chess rules.

stack exec AlphaGambit-exe.exe is run with the following GHC options: -threaded -rtsopts -with-rtsopts=-N -funfolding-use-threshold=16 -O2 -optc-O3

Listing 1. Interface Relevant Chessica Game Details

```haskell
module Functions.States (generateGame, nextStates) where

import Chess
import Chess.Rulebook.Standard (standardRulebook)

generateGame :: Game
generateGame = standardRulebook.newGame

allUpdates :: Game -> Rulebook -> [Update]
allUpdates game rulebook =
  let sameColor (Some (PlacedPiece _ piece)) = piece.color ==
      ↪ game.activePlayer.color
      potentialUpdates (Some (PlacedPiece position _)) = rulebook.updates
         ↪ position game
   in concatMap potentialUpdates $ filter sameColor $ pieces game.board

nextStates :: Game -> [Update]
nextStates game
  | null updates = [Update game endTurn]
  | otherwise = updates
  where updates = allUpdates game standardRulebook
```

2.3.2 Heuristics

Listing 2. Heuristic 1: Piece Point Difference, Heuristic 2: Threat Score

```haskell
{-# LANGUAGE GADTs #-}

module Functions.Scoring (heuristic) where
import Chess
import Chess.Rulebook.Standard.Threat

-- Heuristic 1
heuristic :: Game -> Int
heuristic game = values (piecesOf White game.board) (piecesOf Black game.board)
  where
    values p1 p2 = value p1 - value p2
    value pieces_ = sum [pieceValue piece | piece <- pieces_]
    pieceValue :: Some PlacedPiece -> Int
    pieceValue (Some (PlacedPiece _ (Piece piece _))) = case piece of
      Pawn -> 1
      Knight -> 3
      Bishop -> 3
      Rook -> 5
      Queen -> 9
      King -> 100

foldl' _ z [] = z
foldl' f z (x:xs) = let z' = z `f` x
                    in seq z' $ foldl' f z' xs

-- Heuristic 2
```

```haskell
heuristic :: Game -> Int
heuristic game = value (piecesOf White game.board) - value (piecesOf Black
    ↪ game.board)
  where
    value pieces_ = foldl' (+) 0 [pieceValue piece | piece <- pieces_] +
        ↪ threatScore pieces_
    pieceValue :: Some PlacedPiece -> Int
    pieceValue (Some (PlacedPiece _ (Piece piece _))) =
      case piece of
        Pawn -> 10
        Knight -> 30
        Bishop -> 30
        Rook -> 50
        Queen -> 90
        King -> 1000

    threatScore :: [Some PlacedPiece] -> Int
    threatScore pieces_ = foldl' (+) 0 [length (threats p game.board) | (Some
        ↪ p) <- pieces_]
```

2.3.3 Sequential MiniMax

Listing 3. Sequential MiniMax Algorithm

```haskell
import Control.Parallel.Strategies
import Data.Function (on)
import Data.List (maximumBy, minimumBy)
import Functions.Scoring (heuristic)
import Functions.States (nextStates)

randomCommand :: Game -> Update
randomCommand game =
  let states = nextStates game
   in states !! (length states `div` 2)

findMinTuple :: [(Int, Update)] -> (Int, Update)
findMinTuple = minimumBy (compare `on` fst)

findMaxTuple :: [(Int, Update)] -> (Int, Update)
findMaxTuple = maximumBy (compare `on` fst)

minimax :: Game -> Int -> Player -> Update
minimax game depth player
  | player.color == Chess.Color.White = snd $ findMaxTuple $ map (\update ->
      ↪ (min_ (depth - 1) update.game, update)) (nextStates game)
  | player.color == Chess.Color.Black = snd $ findMinTuple $ map (\update ->
      ↪ (max_ (depth - 1) update.game, update)) (nextStates game)
  where
    max_ :: Int -> Game -> Int
    max_ 0 game = heuristic game
    max_ depth game = maximum (map (\update -> min_ (depth - 1) update.game)
        ↪ (nextStates game))

    min_ :: Int -> Game -> Int
    min_ 0 game = heuristic game
    min_ depth game = minimum (map (\update -> max_ (depth - 1) update.game)
        ↪ (nextStates game))
```

2.3.4 Parallel MiniMax

Listing 4. Parallel MiniMax Algorithm

```
minimaxPar :: Game -> Int -> Player -> Update
minimaxPar game depth player
  | player.color == Chess.Color.White = snd $ findMaxTuple $ map (\update ->
      ↪ (min_ (depth - 1) update.game, update)) (nextStates game)
  | otherwise = snd $ findMinTuple $ map (\update -> (max_ (depth - 1)
      ↪ update.game, update)) (nextStates game)
  where
    max_ :: Int -> Game -> Int
    max_ 0 game1 = heuristic game1
    max_ d game2 = maximum (map (\update -> min_ (d - 1) update.game)
        ↪ (nextStates game2) `using` parList rseq)

    min_ :: Int -> Game -> Int
    min_ 0 game1 = heuristic game1
    min_ d game2 = minimum (map (\update -> max_ (d - 1) update.game)
        ↪ (nextStates game2) `using` parList rseq)
```

2.3.5 Alpha-Beta Pruning

Listing 5. Alpha-Beta Algorithm

```
alphaBeta :: Game -> Int -> Player -> Update
alphaBeta game depth player
  | player.color == Chess.Color.White = snd $ findMaxTuple $ map (\update ->
      ↪ (minValue update.game (depth-1) (-2) 2, update)) (nextStates game)
  | otherwise = snd $ findMinTuple $ map (\update -> (maxValue update.game
      ↪ (depth-1) (-2) 2, update)) (nextStates game)
  where
    maxValue :: Game -> Int -> Int -> Int -> Int
    maxValue g 0 _ _ = heuristic g
    maxValue g d a b =
      let states = reverse $ nextStates g

          getMinimaxAndAlpha :: (Int, Int) -> Update -> (Int, Int)
          getMinimaxAndAlpha (bestMinimaxVal, _) update =
            let newMinimax = max bestMinimaxVal (minValue update.game (d - 1) a
                ↪ b)
             in (newMinimax, max a newMinimax)

          (bestMinimax, _) = takeFirstWithOrLastElem (\(v, _) -> v >= b) $
              ↪ scanl getMinimaxAndAlpha (-2, a) states
       in bestMinimax
    minValue :: Game -> Int -> Int -> Int -> Int
    minValue _ 0 _ _ = heuristic game
    minValue g d a b =
      let states = reverse $ nextStates g

          getMinimaxAndBeta :: (Int, Int) -> Update -> (Int, Int)
          getMinimaxAndBeta (bestMinimaxVal, b) update =
          let newMinimax = min bestMinimaxVal (maxValue update.game (d - 1) a
              ↪ b)
           in (newMinimax, min b newMinimax)

        (bestMinimax, _) =
          takeFirstWithOrLastElem (\(v, _) -> v <= a) $
            scanl getMinimaxAndBeta (2, b) states
     in bestMinimax

-- will take the first element satisfying the condition, or the last element
    ↪ if none do (last wont be checked)
takeFirstWithOrLastElem :: (a -> Bool) -> [a] -> a
takeFirstWithOrLastElem cond [x] = x
takeFirstWithOrLastElem cond (x : xs) = if cond x then x else
    ↪ takeFirstWithOrLastElem cond xs
```

2.3.6 *Jamboree (Parallel Alpha-Beta Pruning)*

Listing 6. Jamboree Algorithm

```
jamboreee :: Game -> Int -> Int -> Int -> Int
jamboreee game _ _ 0 = heuristic game
jamboreee game a b depth | firstVal >= b = firstVal
                         | otherwise = jamboree2 (max firstVal a) b firstVal
  where
    jamboree2 :: Int -> Int -> Int -> Int
    jamboree2 alpha beta bb = maximum (map (\update-> result (-jamboreee
        ↪ update.game (-alpha-1) (-alpha) (depth - 1))) possibleMoves `using`
        ↪ parList rseq )
      where
        result :: Int -> Int
        result res | res >= beta = res
                   | val >= beta = val
                   | otherwise = max (max val res) bb
          where
            val = maximum (map (\update->(-jamboreee update.game (-beta)
                ↪ (-alpha) (depth - 1))) possibleMoves `using` parList rseq)

    firstVal = -jamboreee (head possibleMoves).game (-a) (-b) (depth - 1)
    possibleMoves = nextStates game
```

2.3.7 *Main*

Listing 7. Main

```
module Main (main) where

import Chess
import Chess.Rulebook.Standard (standardRulebook)
import Lib (generateGame, minimax, jamboree, alphaBeta, minimaxPar,
    ↪ randomCommand)
import System.Environment (getArgs, getProgName)
import System.Exit (die)
import System.IO (hSetBuffering, stdout, BufferMode (LineBuffering))

main :: IO ()
main = do
  args <- getArgs
  hSetBuffering stdout LineBuffering
  case args of
    ["sequential", d] -> do
     let game = generateGame
     let minimaxUpdate = minimax game (read d :: Int) game.activePlayer
     putStrLn "Minimax update:"
     print minimaxUpdate.command
    ["parallel", d] -> do
     let game = generateGame
     let minimaxUpdate = minimaxPar game (read d :: Int) game.activePlayer
     putStrLn "Minimax update:"
     print minimaxUpdate.command
    ["jamboree", d] -> do
     let game = generateGame
     let minimaxUpdate = jamboree game (read d :: Int) game.activePlayer
     putStrLn "Jamboree update:"
     print minimaxUpdate.command
    ["alpha_beta", d] -> do
     let game = generateGame
     let minimaxUpdate = alphaBeta game (read d :: Int) game.activePlayer
     putStrLn "Alpha-Beta update:"
     print minimaxUpdate.command
    ["play"] -> do
     let game = generateGame
     playGame game
    _ -> do
     name <- getProgName
     die $ "Usage: " ++ name ++ " <sequential|parallel> <depth=1,2,3,4,5> OR
       ↪ " ++ name ++ " play"

playGame :: Game -> IO ()
playGame g = case standardRulebook.status g of
  Win player -> putStrLn $ "Win for " ++ show player.color
  Draw -> putStrLn "Draw"
  Turn (Player White) -> do
    let (Update game command) = minimaxPar g 4 g.activePlayer
    print game.board
    playGame game
  Turn (Player Black) -> do
    let (Update game command) = minimaxPar g 4 g.activePlayer
    print game.board
    playGame game
```

2.3.8 Visual Representation

Listing 8. Visual Printout

```
import Chess
import Functions.Scoring (heuristic)
import Functions.Search (jamboree, minimax, minimaxPar, randomCommand,
    ↪ alphaBeta)
import Functions.States (generateGame, nextStates)

instance Show Board where
```

```haskell
  show board = unlines [showRow r | r <- [0 .. 7]]
    where
      showRow r = unwords [showPosition (boundedPosition r c) | c <- [0 .. 7]]
      showPosition pos = maybe "." showPiece (Chess.lookup pos board)
      showPiece (Some (PlacedPiece _ piece)) = unicodeChess piece

unicodeChess :: Piece t -> String
unicodeChess (Piece piece color) = case (piece, color) of
  (King, White) -> "U+2654"
  (Queen, White) -> "U+2655"
  (Rook, White) -> "U+2656"
  (Bishop, White) -> "U+2657"
  (Knight, White) -> "U+2658"
  (Pawn, White) -> "U+2659"
  (King, Black) -> "U+265A"
  (Queen, Black) -> "U+265B"
  (Rook, Black) -> "U+265C"
  (Bishop, Black) -> "U+265D"
  (Knight, Black) -> "U+265E"
  (Pawn, Black) -> "U+265F"
```

Listing 9. Unicode to Image

```python
from PIL import Image, ImageDraw

#
    ↪ https://commons.wikimedia.org/wiki/Category:PNG_chess_pieces/Standard_transparent
pieces_mapping = {
    'U+2654': './pieces/white_king.png',
    'U+2655': './pieces/white_queen.png',
    'U+2656': './pieces/white_rook.png',
    'U+2657': './pieces/white_bishop.png',
    'U+2658': './pieces/white_knight.png',
    'U+2659': './pieces/white_pawn.png',
    'U+265A': './pieces/black_king.png',
    'U+265B': './pieces/black_queen.png',
    'U+265C': './pieces/black_rook.png',
    'U+265D': './pieces/black_bishop.png',
    'U+265E': './pieces/black_knight.png',
    'U+265F': './pieces/black_pawn.png',
    '.': './pieces/empty_square.png'
}


def chessboard_to_image(position):
    square_size = 64
    board_size = 8 * square_size

    chessboard = Image.new('RGB', (board_size, board_size), color='white')
    draw = ImageDraw.Draw(chessboard)

    for i in range(8):
        for j in range(8):
            if (i + j) % 2 == 0:
                draw.rectangle([(i * square_size, j * square_size), ((i + 1) *
                    ↪ square_size, (j + 1) * square_size)], fill='green')

    for i, row in enumerate(position.split('\n')):
        for j, piece in enumerate(row.split()):
            piece_image = Image.open(pieces_mapping[piece])
            chessboard.paste(piece_image, (j * square_size, i * square_size),
                ↪ mask=piece_image.split()[3] if piece_image.mode == 'RGBA'
                ↪ else None)

    return chessboard

sample_position = '''
U+2656 U+2658 U+2657 U+2655 U+2654 U+2657 U+2658 U+2656
U+2659 U+2659 U+2659 U+2659 . U+2659 U+2659 U+2659
. . . . U+2659 . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .

U+265C U+265E U+265D U+265B U+265A U+265D U+265E U+265C
'''

# chessboard_image = chessboard_to_image(sample_position)
# chessboard_image.save('chessboard_image_with_filled_grid.jpg')

with open('../output.txt', 'r', encoding='utf-8') as file:
    contents = file.read()


boards = contents.split('\n\n')


for i, board in enumerate(boards):
    chessboard_image = chessboard_to_image(board)
    chessboard_image.save(f'./board_states/chessboard_{i + 1}.png')
    if i > 100:
        break
```

2.3.9 Testing and Performance Evaluation

Listing 10. Timing

```
#!/bin/bash

# Define the Haskell executable name
executable="AlphaGambit-exe"

# Function to run the executable with given parameters and append to the
    ↪ appropriate CSV file
run_and_record() {
    mode=$1
    depth=$2
    csv_file=$3

    # Using 'time' to measure the execution time
    start_time=$(gdate +%s.%N)
    stack exec --silent $executable $mode $depth
    end_time=$(gdate +%s.%N)

    # Calculate duration
    duration=$(echo "$end_time - $start_time" | bc)

    # Append to the respective CSV file
    echo "$mode,$depth,$duration" >> $csv_file
}

# Headers for CSV files

modes=("parallel sequential alpha_beta jamboree")

# for mode in $modes; do
# echo "mode,depth,duration" > new_runtimes/${mode}.csv
# done

# Run the executable once to warm it up
stack exec --silent $executable parallel 1

# Loop through all combinations of mode and depth
for mode in $modes; do
    for depth in {1..5}; do
        run_and_record $mode $depth new_runtimes/${mode}.csv
    done
done
```

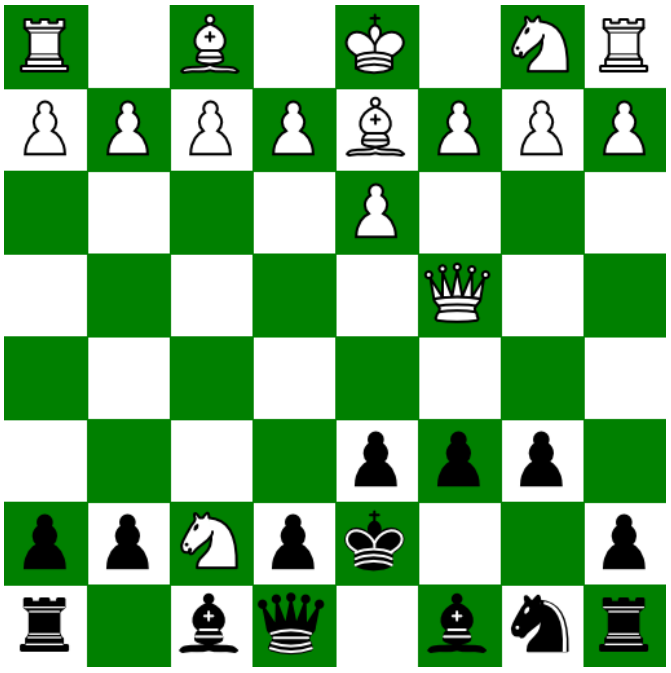


Figure 1. An example image of a Chess Board State converted to an Image in Color

3 RESULTS

3.1 Performance Improvements

Table 1. Performance Comparison (Seconds)

Depth	Version			
	Sequential	Parallel	Alpha-Beta	Jamboree
1	0.46073360	0.44102400	0.43213000	0.43370427
2	0.45661260	0.45061940	0.43771255	0.43347273
3	0.62103900	0.49432600	0.49344282	0.45055836
4	4.20282800	1.21160080	0.91326627	0.52318300
5	91.26088060	18.10007400	1.91300927	1.25586491

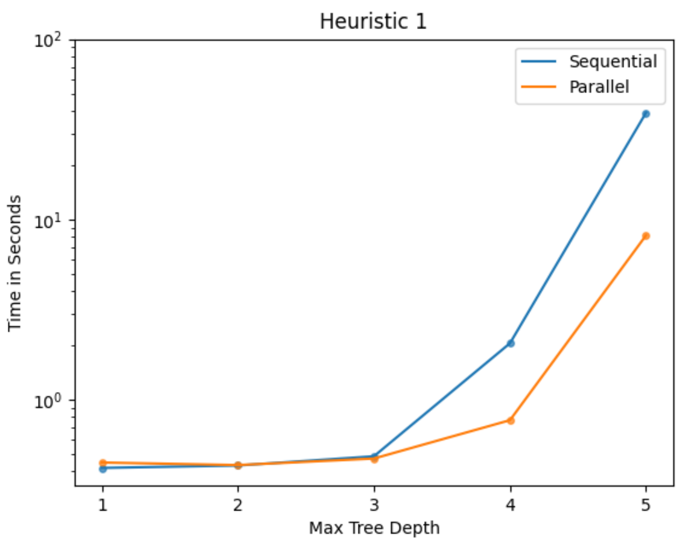


Figure 2. Heuristic 1 Performance for Sequential and Parallel MiniMax

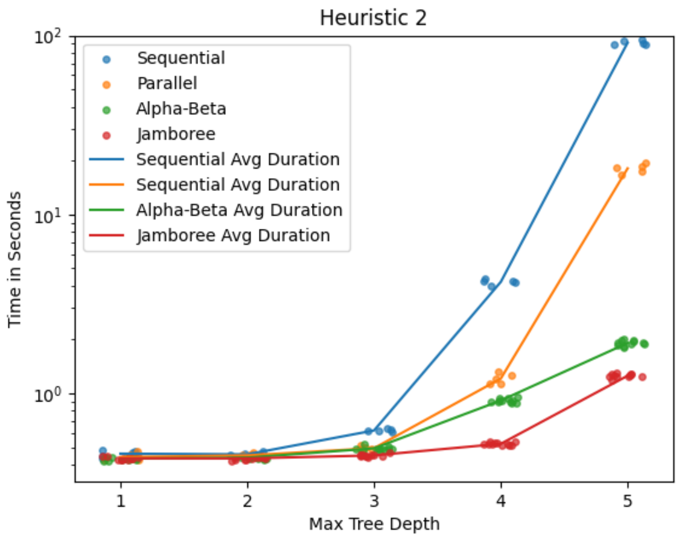


Figure 3. Heuristic 2 Performance for All Four Algorithms

Parallel minimax performs exponentially better than sequential minimax as seen in the log scale graph comparisons in Figures 1 and 2. However, we see that the more advanced algorithms like Alpha-Beta and Jamboree are still exponentially faster than Parallel Minimax.

3.2 Parallelism Analysis

Below is the timing values for parallel functions with the indicated number of cores running at depth 4.

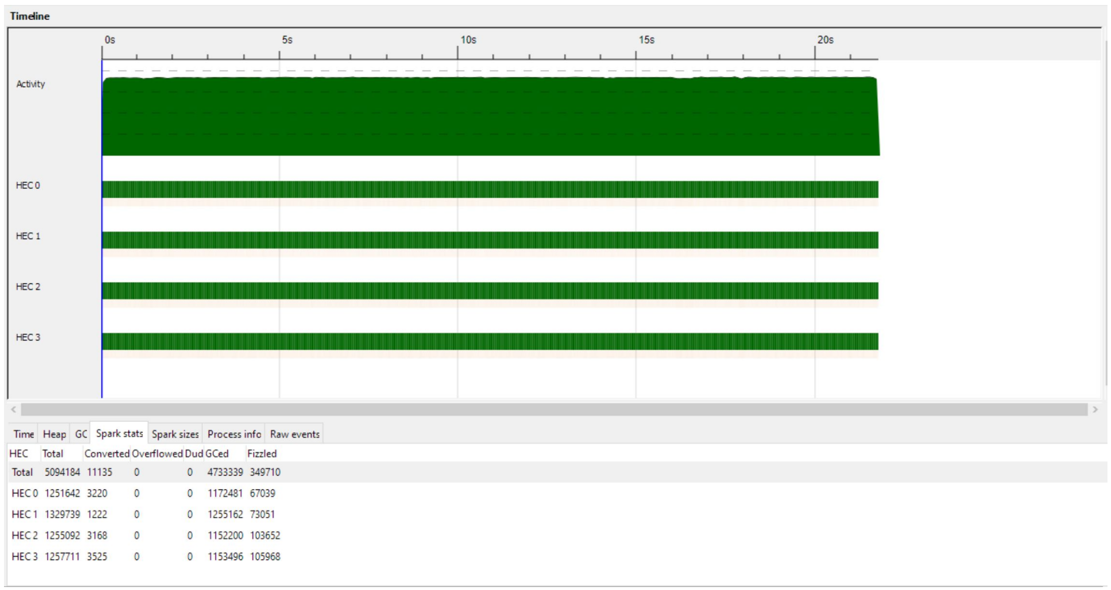
Table 2. Core Performance Comparison (Seconds)

Cores	Version
	Parallel Minimax
1	3.578
8	1.181
16	0.847

In an ideal parallelization scenario, each 4 cores should offer a 4 times speedup compared to 1 core. In our case, we see the increase in cores results in a 300% speedup. However as the number of cores increases the less efficient the process becomes, with two times the number of cores resulting in a 35% speedup.

3.3 Threadscope analysis

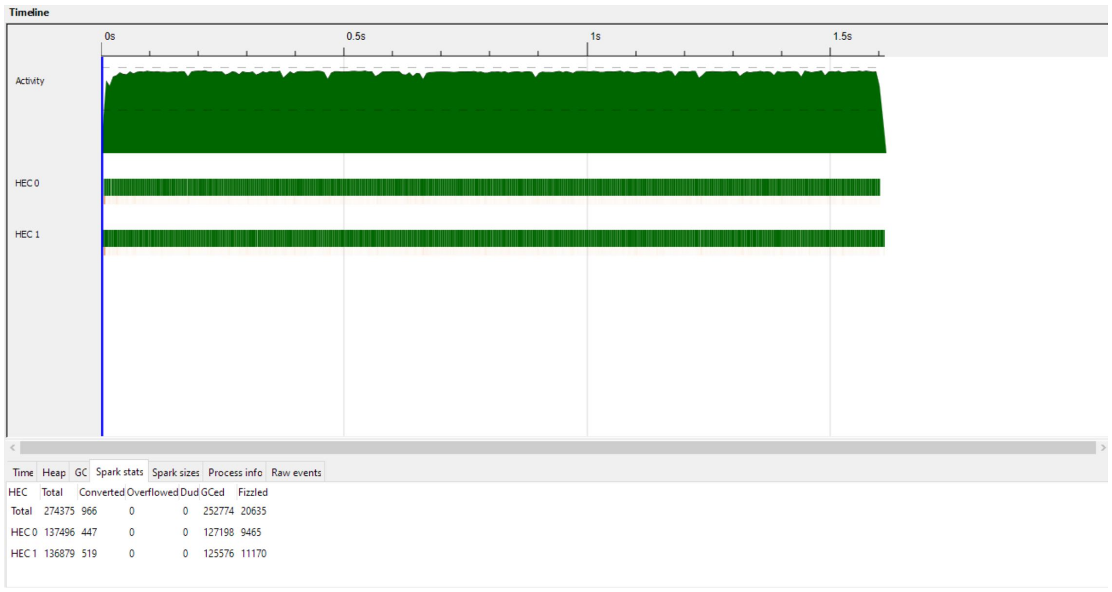
Threadscope data for parallel minimax:



HEC	Total	Converted	Overflowed	Dud	GCed	Fizzled
Total	5094184	11135	0	0	4733339	349710
HEC 0	1251642	3220	0	0	1172481	67039
HEC 1	1329739	1222	0	0	1255162	73051
HEC 2	1255092	3168	0	0	1152200	103652
HEC 3	1257711	3525	0	0	1153496	105968

Figure 4. ThreadScope Data for Parallel Minimax with depth 5

Threadscope data for Jamboree:



HEC	Total	Converted	Overflowed	Dud	GCed	Fizzled
Total	274375	966	0	0	252774	20635
HEC 0	137496	447	0	0	127198	9465
HEC 1	136879	519	0	0	125576	11170

Figure 5. ThreadScope Data for Jamboree with depth 5

We see that the work is split well between the 4 cores in this example, meaning that parallelism is speeding up our calculations. The Garbage Collection time is low, however, there are a lot of sparks being GC'd or fizzled because they are unused. This indicates the program still be made more efficient through parallelization.

3.4 Simulated Game Run

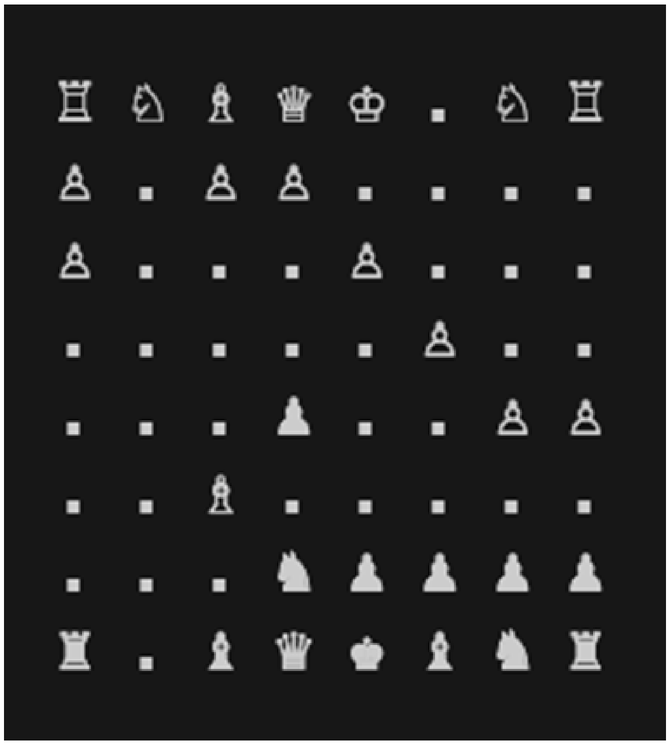


Figure 6. An example image of a Chess Board State Outputted from Haskell

4 CONCLUSION

4.1 Challenges Faced

4.1.1 Integration Issues with Chessica

Chessica's lack of comprehensive documentation means there is a steep learning curve when first adopting the package. Its lack of functions for inputting and outputting game states, also makes it harder to debug and test. However, Chessica was still the most complete package we could find encoding chess rules in Haskell, and it worked well outside of these limitations.

4.1.2 Inefficient Parallelism

Using ParList directly on Minimax/Jamboree would create a lot of sparks that were GC'd or fizzled, meaning they didn't contribute to the program speed up. We believe the issue to be that the processes towards the nodes of the trees run too fast, so the values are already computed in the main thread and not used, therefore fizzling or being discarded. We attempted to address this issue using parListChunk, which would split the array into equal sections and call parList. Since most of the computation is getting the array elements themselves, this theoretically would result in an improvement due to more sparks being evaluated. However, we found that this change made no difference. Additionally, we tried using rdeepseq instead of rseq to forcibly evaluate values, but that did not show any improvements either.

4.1.3 Complexities in Parallelizing Jamboree

The parallelization of Jamboree, a critical component of the system, proved to be tough because it requires early stopping upon a certain branch condition. Because this was hard to achieve in the parallel setting, we developed a workaround where all parallel operations had to conclude before proceeding, consequently impacting computational speed. This results in the parallelized version being similar or worse in time compared to normal Alpha-Beta.

4.2 Potential Improvements

4.2.1 Optimizations of Functions

Further code refinement could enhance the performance of all search functions within the implemented system: streamlining and augmenting the efficiency of the search algorithms would increase the speed and accuracy of decision-making processes.

In terms of parallelism, this would include ensuring lazy evaluation so that sparks can be handed off to other cores without being destroyed/unused.

4.2.2 Enhanced Parallelism Deconstruction for Jamboree

Deconstruction and reconfiguring the parallelism structure could also enhance performance: fine-tuning the parallelization strategy could improve the efficiency of Jamboree, potentially optimizing computational speed.

4.2.3 Augmentation of Heuristic Functionality

Broadening the heuristic function to consider aspects like piece mobility and safety would lead to a more comprehensive evaluation of board states, refining the decision-making process.

4.2.4 Exploration of Complex Algorithms - Monte Carlo Tree Search

The integration and exploration of more intricate algorithms, such as the Monte Carlo Tree Search (MCTS), present a compelling avenue for system enhancement. The adoption of MCTS, renowned for its ability to navigate complex decision spaces, holds the potential to broaden the system's repertoire and augment its capability to analyze and derive optimal moves in intricate chess scenarios.

REFERENCES

[bri] ”chess - game”. *Encyclopaedia Britannica*.

[2] chessprograming (2021). Jamboree.

[3] Haskell (2023a). ”chessica - haskell chess library”. *Haskell Hackage*.

[4] Haskell (2023b). ”parallelism and concurrency in haskell”. *Haskell Wiki*.

[5] Hooper, David, K. W. (1992). The oxford companion to chess. *Oxford University Press*.

[6] Mozolewski, M. (2020). Minmax, alpha beta pruning; a*, dfs bfs in haskell.

[7] Murray, H. J. R. (1913). A history of chess. *Oxford University Press*.

[8] Newborn, M. (1997). ”kasparov versus deep blue: Computer chess comes of age.”.