Overview

Background:
For our project, we decided to create a bot capable of playing checkers against a user.
We use a 8x8 board, with each player starting with 12 pieces arranged on the three rows closest to
them in the "straight checkers" starting position. The goal is to either capture all of the opponent’s
pieces or block their moves, and the game ends when a player cannot make any more moves or
is out of pieces. Pieces begin as pawns and can capture opponents by jumping over them; they
can only move diagonally forward. A pawn becomes a king when it reaches the opposite end of
the board, and kings can move diagonally backward, capturing pieces int he same way. As starter
code for our implementation, we modified existing code that allowed for two users to play against
each other (see references). We clearly designated our additions.

Initial Goal:
Our initial goal was to create a bot that played against the user. However, we found that parallel-
lezing this properly was a bit difficult given that the bot had to wait for user input.

Modified Goal:
Instead, we decided to simulate a game of checkers using two bots, a Red bot and a Black bot.
This allowed for better parallellization, as there was no wait for user input, and the game can run
start to finish independently.
**Setup:**

---

Welcome to our CheckerBot!
-------------------------------
If your valid moves are listed as [], you have tried to move a piece you cannot. In that situation, please just use that same input when prompted for where you want to move the piece.

---

Game Begins: Red Starts
1  b  b  b  b  b  b
2  b  b  b  b  b  b
3  b  b  b  b  b  b
4  ---------
5  r  r  r  r  r  r
6  r  r  r  r  r  r
7  r  r  r  r  r  r
8  r  r  r  r  r  r

Red's Turn:
Red Chooses: Move (5,6) (4,8)
Heuristic Value after Red's Move: 59
1  b  b  b  b  b  b
2  b  b  b  b  b  b
3  b  b  b  b  b  b
4  ---------
5  r  r  r  r  r  r
6  r  r  r  r  r  r
7  r  r  r  r  r  r
8  r  r  r  r  r  r

Black's Turn:
Black Chooses: Move (9,3) (5,4)
Heuristic Value after Black's Move: 942
1  b  b  b  b  b  b
2  b  b  b  b  b  b
3  b  b  b  b  b  b
4  ---------
5  r  r  r  r  r  r
6  r  r  r  r  r  r
7  r  r  r  r  r  r
8  r  r  r  r  r  r

---

**How our bots operate**

Each bot uses minimax with heuristics that prioritize moves based on if they allow for a capture of an opponent piece, how far down the board the piece is after that move (closer to being a king), and if the move allows for the piece to become a king. They maximize their own score and minimize their opponent’s score. We played around with heuristics (i.e. having each use different weights and heuristic values), but as both are using minimax, the starting player (Red) has a clear advantage, which is seen in the outcomes of the simulations. We also attempted to find existing implementation of a Haskell checkers bot, but this led to the same results.

**How we attempted to parallelize**

Our initial goal was to parallelize a fully interactive game between bot and user. However, having that first game coded and then evaluated on threadscope, it was clear that user input would affect our performance metrics. As the bot waited for the user’s move, execution time was disrupted. We concluded that our project would be improved if transformed into a bot vs bot game, that is, fully automated. As we expected, this change allowed us to work with more predictable metrics. This was essential for properly calculating speedup and for a controlled game development.

Our first attempt to parallelize used parMap by parallelizing for every depth. Although this approach gave us great speedup results, our number of sparks was reaching a total of 3,000,000 created and of those 1% was converted, for a total of around 105,000 sparks converted. The majority of sparks were being garbage collected.

We shifted to a second parallelization approach. In order to avoid the unnecessary creation of sparks, we changed our code so that parallelizing would only occur for depth \( \geq 4 \). This lead to a considerable improvement in the performance metrics of our game. Total number of sparks went down to 6,000 and of those 85% were converted, for a total of around 5,000 sparks converted.
Final parallelization stats

Our code ultimately ran at 24.886 second for 1 core, and a second core brought that time down to 13.626 seconds. Our actual ratio for a second core, as seen in the graph below, is close to the ideal ratio (around 12.443 seconds).
<table>
<thead>
<tr>
<th># Threads</th>
<th>Actual Runtime (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>24.886</td>
</tr>
<tr>
<td>2</td>
<td>13.626</td>
</tr>
<tr>
<td>3</td>
<td>11.55</td>
</tr>
<tr>
<td>4</td>
<td>10.542</td>
</tr>
<tr>
<td>5</td>
<td>9.286</td>
</tr>
<tr>
<td>6</td>
<td>9.706</td>
</tr>
<tr>
<td>7</td>
<td>9.185</td>
</tr>
<tr>
<td>8</td>
<td>8.844</td>
</tr>
<tr>
<td>9</td>
<td>9.302</td>
</tr>
<tr>
<td>10</td>
<td>9.2012</td>
</tr>
<tr>
<td>11</td>
<td>8.849</td>
</tr>
<tr>
<td>12</td>
<td>10.65</td>
</tr>
</tbody>
</table>

Code (Main.hs):

Note that sections taken from the aforementioned source code are clearly designated.

```hs
import Control.Parallel.Strategies
import Data.List (maximumBy)

--- Slightly modified data structures from source code to be Black and Red
data CPiece = Red | RedKing | Black | BlackKing deriving (Eq, Show)
type CBoard = [[Maybe CPiece]]
data CMove = Move (Int,Int) (Int, Int) | Take (Int,Int) (Int, Int) deriving (Eq, Show)

{-
Not in source code:
-}

Functions:
pieceWeight, positionWeight, getHeuristicValueRed, evaluateBoardRed,
generateHeuristicValueBlack, evaluateBoardBlack

pieceWeight :: CPiece -> Int
pieceWeight Red = 1
pieceWeight RedKing = 10
pieceWeight Black = 1
pieceWeight BlackKing = 10

positionWeight :: (Int, Int) -> Int
positionWeight (_, y) = y

getHeuristicValueRed :: CBoard -> Int
getHeuristicValueRed board = evaluateBoardRed board

evaluateBoardRed :: CBoard -> Int
evaluateBoardRed board = redScore - blackScore + captureBonus

where
  redScore = sum [pieceWeight p * positionWeight (x, y) | x <- [1..8],
                           y <- [1..8], Just p <- getPiece board (x, y), isRedPiece (Just p)]
  blackScore = sum [pieceWeight p * positionWeight (x, y) | x <- [1..8],
                          y <- [1..8], Just p <- getPiece board (x, y), isBlackPiece (Just p)]
  captureBonus = sum [captureValue | possibleMove <- allPossibleMoves board, isCaptureMove possibleMove]

  captureValue = 1000

  isCaptureMove :: CMove -> Bool
  isCaptureMove (Take _ _) = True
  isCaptureMove _ = False

  allPossibleMoves :: CBoard -> [CMove]
  allPossibleMoves b = concatMap (| pos | listMoves b pos | allPositions)
allPositions = [(x, y) | x <- [1..8], y <- [1..8]]
```
getHeuristicValueBlack :: CBoard -> Int
getHeuristicValueBlack board = evaluateBoardBlack board

evaluateBoardBlack :: CBoard -> Int
evaluateBoardBlack board = blackScore + redScore + captureBonus
where
  redScore = sum [pieceWeight p * positionWeight (x, y) |
                  x <- [1..8], y <- [1..8], Just p <- [getPiece board (x, y)], isRedPiece (Just p)]
  blackScore = sum [pieceWeight p * positionWeight (x, y) |
                    x <- [1..8], y <- [1..8], Just p <- [getPiece board (x, y)], isBlackPiece (Just p)]
  captureBonus = sum [captureValue | possibleMove <- allPossibleMoves board, isCaptureMove possibleMove]

captureValue = 1000

isCaptureMove :: CMove -> Bool
isCaptureMove (Take _ _) = True
isCaptureMove _ = False

allPossibleMoves :: CBoard -> [CMove]
allPossibleMoves b = concatMap (pos => listMoves b pos) allPositions
allPositions = [(x, y) | x <- [1..8], y <- [1..8]]

{-
   From source code, slight modifications to Red and Black
-}

Functions:
makeEmptyBoard, makeStandardBoard, move, movePiece, listMoves, getPiece
updatePiece, setPiece, width, height, allMoves, allTakes, printBoard
-

makeEmptyBoard :: Int -> Int -> CBoard
makeEmptyBoard w h = [[Nothing | _ <- [1..w]] | _ <- [1..h]]

makeStandardBoard :: CBoard
makeStandardBoard = board where
  board = foldl (x y => setPiece x (Just Black) y) w_board b_positions
  w_board = foldl (x y => setPiece x (Just Red) y) initial_w_positions
  b_positions = [(x, y) | x <- [1..8], y <- [1..8], (x `mod` 2)`/=`(y `mod` 2)]
  initial_w_positions = [0,1,2,3] /
     [6,7,8] /
     [x `mod` 2)`/=`(y `mod` 2)

move :: CBoard -> CMove -> Maybe CBoard
move b m =if any (==m) (listMoves b (x1,y1))
            then Just (movePiece b m) else Nothing
        else if any (==m) (listMoves b (x1,y1))
            then Just (movePiece b m) else Nothing

movePiece :: CBoard -> CMove -> CBoard
movePiece b (Move (x1, y1) (x2, y2)) = setPiece b' p (x2, y2) where
  p = updatePiece b (x1, y1) (x2, y2)
  b' = setPiece b (Nothing) (x1, y1)
movePiece b (Take (x1, y1) (x2, y2)) = setPiece b'' p (x2, y2) where
  p = updatePiece b (x1, y1) (x2, y2)
  b'' = setPiece b' (Nothing) (quot (x1-x2) 2, quot (y1-y2) 2)
  b' = setPiece b (Nothing) (x1, y1)

listMoves :: CBoard -> (Int,Int) -> [CMove]
listMoves b (x, y) = case getPiece b (x, y) of
  Just Black -> [Move(x,y) (x2,y2) | (x2,y2) <- allMoves b (x,y), y2>y]
    ++ [Take(x,y) (x2,y2) | (x2,y2) <- allTakes b (Just Black) (x,y)]
  Just Red -> [Move(x,y) (x2,y2) | (x2,y2) <- allMoves b (x,y), y2>y]
    ++ [Take(x,y) (x2,y2) | (x2,y2) <- allTakes b (Just Red) (x,y)]
  Just BlackKing -> [Move(x,y) (x2,y2) | (x2,y2) <- allMoves b (x,y)] ++
    [Take(x,y) (x2,y2) | (x2,y2) <- allTakes b (Just BlackKing) (x,y)]
  Just RedKing -> [Move(x,y) (x2,y2) | (x2,y2) <- allMoves b (x,y)] ++
    [Take(x,y) (x2,y2) | (x2,y2) <- allTakes b (Just RedKing) (x,y)]
Nothing -> []

getPiece :: CBoard -> (Int,Int) -> Maybe CPiece
getPiece board (x, y) = case (getElement board y) of
  Nothing -> Nothing
  Just ls -> case (getElement ls x) of
    Nothing -> Nothing
    Just x -> x

updatePiece :: CBoard -> (Int,Int) -> (Int,Int) -> Maybe CPiece
updatePiece b (x1, y1) (x2, y2) = case getPiece b (x1, y1) of
  Just Black -> if y2 == height b then Just BlackKing else Just Black
  Just Red -> if y2 == 1 then Just RedKing else Just Red
  a -> a

setPiece :: CBoard -> Maybe CPiece -> (Int,Int) -> CBoard
setPiece [] _ (_) = []
setPiece [(x:l):ls] p (w:1) = case (setPiece (l:ls) p ((w-1),1)) of
  (l2:ls2) -> (x:l2):ls2
  [] -> [x]
setPiece (l:ls) p (w,h) = l : (setPiece ls p (w,(h-1)))

width :: CBoard -> Int
width [] = 0
width (h:ht) = length h

height :: CBoard -> Int
height = length
getElem :: [a] -> Int -> Maybe a
getElem [] _ = Nothing
getElem (h:_) 1 = Just h
getElem (h:t) n = getElem t (n-1)

allMoves :: CBoard -> (Int,Int) -> [(Int,Int)]
allMoves b (x,y) = [p | p<-onBoard, (getPiece b p) == Nothing] where
  onBoard = [(x,y) | (x,y)<-spaces, x<width b, y<height b, x>0, y>0]
  spaces = [(x+1,y+1), (x-1,y+1), (x+1,y-1), (x-1,y-1)]

allTakes :: CBoard -> Maybe CPiece -> (Int,Int) -> [(Int,Int)]
allTakes b Nothing _ = []
allTakes b (Just Black) (x,y) = takes where
  takes = [(x2,y2) | (x2,y2)<-free, 
    getPiece b (quot (x2+x) 2, quot (y2+y) 2) == Just Red ||
    getPiece b (quot (x2+x) 2, quot (y2+y) 2) == Just RedKing]
  free = [s | s<-onBoard, (getPiece b s) == Nothing]
  onBoard = [(x,y) | (x,y)<-spaces, x<width b, y<height b, x>0, y>0]
  spaces = [(x+2,y+2), (x-2,y+2), (x+2,y-2), (x-2,y-2)]

allTakes b (Just BlackKing) (x,y) = takes where
  takes = [(x2,y2) | (x2,y2)<-free, 
    getPiece b (quot (x2+x) 2, quot (y2+y) 2) == Just Red ||
    getPiece b (quot (x2+x) 2, quot (y2+y) 2) == Just RedKing]
  free = [s | s<-onBoard, (getPiece b s) == Nothing]
  onBoard = [(x,y) | (x,y)<-spaces, x<width b, y<height b, x>0, y>0]
  spaces = [(x+2,y+2), (x-2,y+2), (x+2,y-2), (x-2,y-2)]

allTakes b (Just Red) (x,y) = takes where
  takes = [(x2,y2) | (x2,y2)<-free, 
    getPiece b (quot (x2+x) 2, quot (y2+y) 2) == Just Black ||
    getPiece b (quot (x2+x) 2, quot (y2+y) 2) == Just BlackKing]
  free = [s | s<-onBoard, (getPiece b s) == Nothing]
  onBoard = [(x,y) | (x,y)<-spaces, x<width b, y<height b, x>0, y>0]
  spaces = [(x+2,y+2), (x-2,y+2), (x+2,y-2), (x-2,y-2)]

allTakes b (Just RedKing) (x,y) = takes where
  takes = [(x2,y2) | (x2,y2)<-free, 
    getPiece b (quot (x2+x) 2, quot (y2+y) 2) == Just Black ||
    getPiece b (quot (x2+x) 2, quot (y2+y) 2) == Just BlackKing]
  free = [s | s<-onBoard, (getPiece b s) == Nothing]
  onBoard = [(x,y) | (x,y)<-spaces, x<width b, y<height b, x>0, y>0]
  spaces = [(x+2,y+2), (x-2,y+2), (x+2,y-2), (x-2,y-2)]

printBoard :: CBoard -> IO ()
printBoard b = putStrLn (showBoard b)
```haskell
-- Modified from source code to be a better representation of the board
showBoard :: CBoard -> String
showBoard board = indexedBoard
  where
    indexedBoard = unlines $ zipWith (\i row -> pad i ++ " | " +
    -- concatMap showPieceWithSpace row ++ "|") [1..] board

-- Not in source code
pad :: Int -> String
pad n |
      n < 10 = " " ++ show n
      otherwise = show n

showPieceWithSpace :: Maybe CPiece -> String
showPieceWithSpace piece = showPiece piece ++ " "

-- Modified from source code
showPiece :: Maybe CPiece -> String
showPiece (Just Red) = "r"
showPiece (Just RedKing) = "R"
showPiece (Just Black) = "b"
showPiece (Just BlackKing) = "B"
showPiece (Nothing) = "-

gameStart :: IO ()
gameStart = do
  putStrLn "***************************************************************************"
  putStrLn "Welcome to ourCheckerBot!"
  putStrLn "***************************************************************************"
  putStrLn "If your valid moves are listed as [], you have tried to move a piece you cannot."
  putStrLn "In that situation, please just use that same input when prompted for where you"
  putStrLn "want to move the piece."
  putStrLn "***************************************************************************"
  putStrLn "Game Begins: Red Starts"
printBoard makeStandardBoard
```
{-
Not in source code

Functions:
getAllBlackPieces, getAllRedPieces, isBlackPiece, isRedPiece, playGame, areMovesAvailable,
minimaxRed, maximizeRed, minimizeRed, minimaxBlack, maximizeBlack, minimizeBlack,
endGameScore, performRedMove, performBlackMove, isGameOver
-

getAllBlackPieces :: CBoard -> [(Int, Int)]
getAllBlackPieces board = [(x, y) | x <- [1..8], y <- [1..8], isBlackPiece (getPiece board (x, y))]

getAllRedPieces :: CBoard -> [(Int, Int)]
getAllRedPieces board = [(x, y) | x <- [1..8], y <- [1..8], isRedPiece (getPiece board (x, y))]

isBlackPiece :: Maybe CPiece -> Bool
isBlackPiece (Just Black) = True
isBlackPiece (Just BlackKing) = True
isBlackPiece _ = False

isRedPiece :: Maybe CPiece -> Bool
isRedPiece (Just Red) = True
isRedPiece (Just RedKing) = True
isRedPiece _ = False

playGame :: CBoard -> IO ()
playGame board = do
  putStrLn "Red's Turn!"
  redMovesAvailable <- areMovesAvailable board Red
  if not redMovesAvailable
     then putStrLn "Game Over! Red has no moves left. Black wins!" >>= return ()
     else do
           redBoard <- performRedMove board
           printBoard redBoard
           blackMovesAvailable <- areMovesAvailable redBoard Black
           if not blackMovesAvailable
              then putStrLn "Game Over! Black has no moves left. Red wins!" >>= return ()
              else do
                    putStrLn "Black's Turn!"
                    blackBoard <- performBlackMove redBoard
                    printBoard blackBoard
                    playGame blackBoard
areMovesAvailable :: CBoard -> CPiece -> ID -> Bool
areMovesAvailable board player =
  return (not . null $ concatMap (\pos -> listMoves board pos) positions)
where
  positions = case player of
    Red -> getAllRedPieces board
    Black -> getAllBlackPieces board
    _ -> []

minimizeRed :: CBoard -> Int -> CMove
minimizeRed board depth = bestMove where
  (_, bestMove) = maximizeRed depth board

maximizeRed :: Int -> CBoard -> Int, CMove
maximizeRed depth board = (evaluateBoardRed board, Move (0, 0) (0, 0))
maximizeRed depth board
  | depth <= 4 = maximumBy \((score1, _) -> compare score1 score2\)
  |   $ parMap rpar \move -> (minimizeRed (depth - 1) (movePiece board move), move) redMoves
  | otherwise = maximumBy \((score1, _) -> compare score1 score2\)
  |   $ map \(move -> minimizeRed (depth - 1) (movePiece board move), move\) redMoves
where
  redMoves = concatMap \((x, y) -> listMoves board (x, y)) $ getAllRedPieces board

minimizeRed :: Int -> CBoard -> Int
minimizeRed depth board = evaluateBoardRed board
minimizeRed depth board
  | depth <= 4 = minimum \$ parMap rpar \move -> fst $ maximizeRed (depth - 1) (movePiece board move) blackMoves
  | otherwise = minimum \$ map \(move -> fst $ maximizeRed (depth - 1) (movePiece board move)) blackMoves
where
  blackMoves = concatMap \((x, y) -> listMoves board (x, y)) $ getAllBlackPieces board

minimizeBlack :: CBoard -> Int -> CMove
minimizeBlack board depth = bestMove where
  (_, bestMove) = maximizeBlack depth board

maximizeBlack :: Int -> CBoard -> Int, CMove
maximizeBlack depth board = (evaluateBoardBlack board, Move (0, 0) (0, 0))
maximizeBlack depth board
  | depth <= 4 = maximumBy \((score1, _) -> compare score1 score2\)
  |   $ parMap rpar \move -> (minimizeBlack (depth - 1) (movePiece board move), move) blackMoves
  | otherwise = maximumBy \((score1, _) -> compare score1 score2\)
  |   $ map \(move -> minimizeBlack (depth - 1) (movePiece board move), move\) blackMoves
where
  blackMoves = concatMap \((x, y) -> listMoves board (x, y)) $ getAllBlackPieces board
References

Source code: https://github.com/IAmSam42/CS312-Checkers
Consulted with: Sanjay Rajasekharan