Minimax Mancala

Chance Onyiorah cco2134

Parallel Functional Programming

Fall 2022

Dec 21, 2022

I. The Game

Mancala is a two-player game with the goal to capture as many marbles as possible. A wooden board contains two rows of six holes each and two pockets on either side called mandalas that are used to store the marbles for each player as shown in Figure 1. The game starts with 6 marbles in each of the six holes. Players take turns choosing a hole on their side and distributing the marbles to the holes in a counter-clockwise direction, making sure to drop a marble in their designated mancala as they pass it. If the last marble lands on the opposing player's side, their turn ends. If it lands in an empty pocket on their own side and there is at least one marble in the hole directly across from it, the player gets to capture both holes' marbles. If the last marble lands in their store, they get to choose another hole. The game ends when either player has an empty row. Any marbles that are not captured at this time, go to the player whose side they were left on.

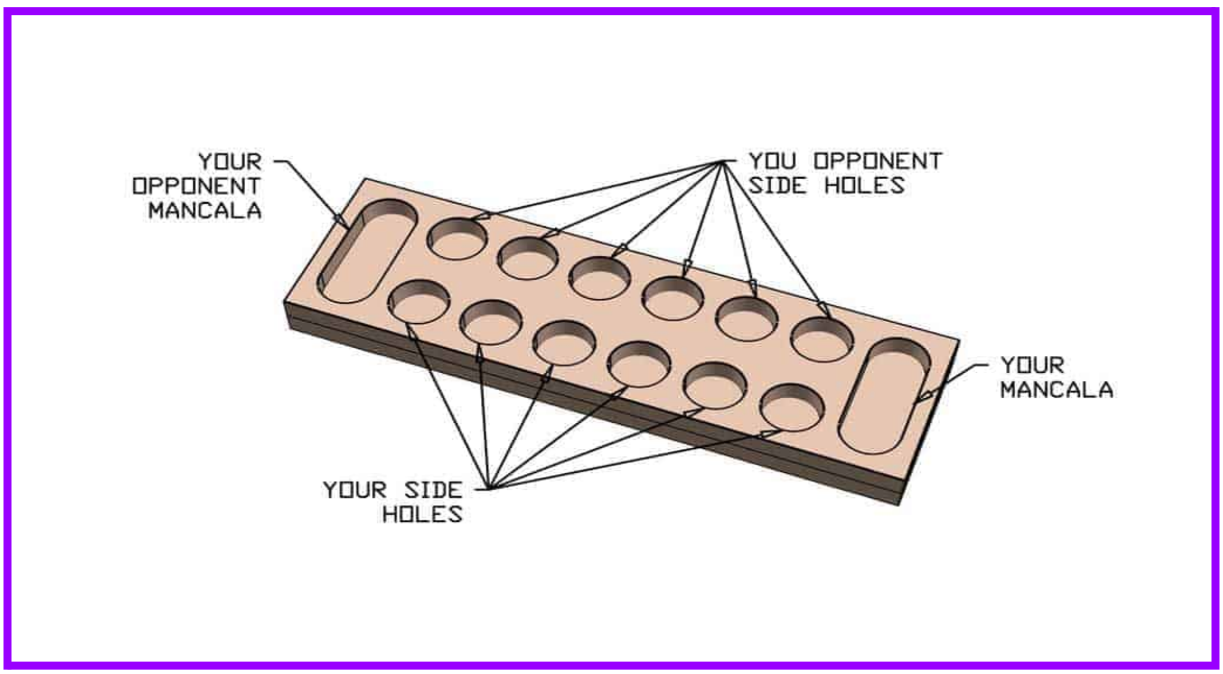


Figure 1. Diagram of Mancala game board

II. Minimax

Minimax is a search algorithm usually used in game-solving to find the best next move. The algorithm works by using a minimizer and a maximizer where the player tries to minimize and maximize their score respectively [1]. This score is calculated by evaluating the current game state and determining which new game state will minimize/maximize the score based on possible moves.

A tree with nodes of next possible games states and evaluates those nodes to see which move has the highest evaluation score (Figure 2). Because there are so many possible moves, a depth limit is passed in to indicate how far down the tree we want to search [1].

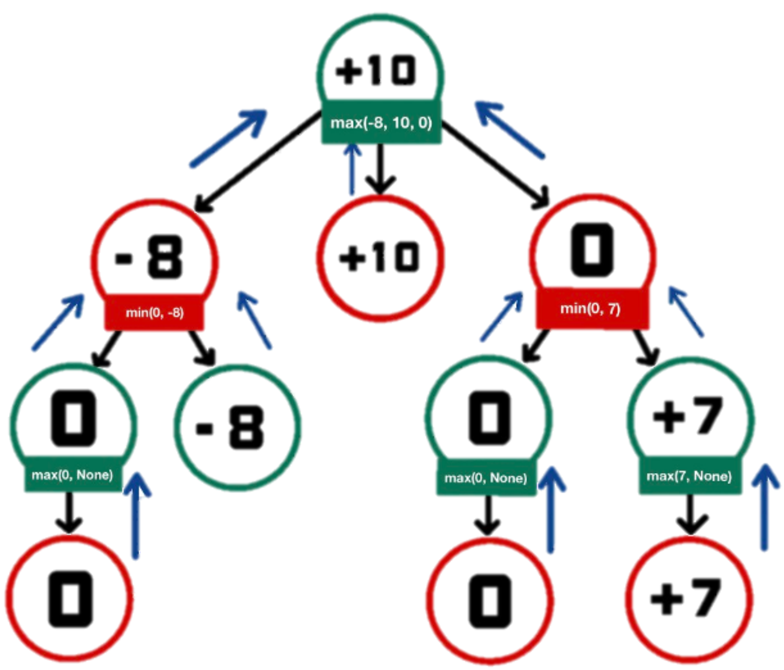


Figure 2. Steps in example minimax tree

III. Alphabeta Pruning

The alphabeta technique is used to optimize the Minimax algorithm. Using this technique, minimax is able to search the nodes of the tree faster. The algorithm will know not to search certain branches of the tree because it will not hold the minimal or maximum value [2]. It works by passing in alpha and beta values to the Minimax algorithm. Alpha is the maximized value and beta is the minimized value. The maximizer updates the alpha value to the maximum value found so far while the minimizer updates the beta value to the minimum value found [3].

IV. Implementation and Parallelism

Below is a snippet of the sequential minimax implementation. The function takes in the current game state, a boolean indicating whether or not we want to minimize or maximize the score, a starting depth, and the depth limit of the tree. It then returns a tuple of the score which is a result of the game board evaluation and the best move to get that score (which is represented by a number on the board).

```
minimax    :: (GameState a) => a -> Bool -> Int -> Int -> (Int, Maybe Int)
minimax gs _ depth depthlimit | depth == depthlimit || gameOver gs = (evaluate gs, Nothing)
minimax gs minimize depth depthlimit =
    let minOrMax = (if minimize then minimumBy else maximumBy) (comparing fst)
        possibilities = (possibleMoves gs)
        scores = map fst $ map (\poss -> (minimax (makePossibility gs poss) (not minimize) (depth+1) depthlimit)) possibilities
        wrappedPossibilities = map Just possibilities
        scorePossPairs = zip scores wrappedPossibilities in
    minOrMax scorePossPairs
```

The parallelized minimax algorithm is similar to the sequential, except that it implements parList from the `Control.Parallel.Strategies` library to evaluate each list element in parallel based on a given strategy. A strategy takes a data structure as input and parallelizes it using rpar and rseq to then return the original value [4]. In this case, the strategy is rseq which evaluates an argument to its Weak Head Normal Form (WHNF) [4]. WHNF is defined as when the outermost part has been evaluated to the lambda abstraction [5].

```
minimaxPar    :: (GameState a) => a -> Bool -> Int -> Int -> (Int, Maybe Int)
minimaxPar gs _ depth depthlimit | depth == depthlimit || gameOver gs = (evaluate gs, Nothing)
minimaxPar gs minimize depth depthlimit =
    let minOrMax = (if minimize then minimumBy else maximumBy) (comparing fst)
        possibilities = (possibleMoves gs)
        scores = (map fst $ map (\poss -> (minimaxPar (makePossibility gs poss) (not minimize) (depth+1) depthlimit)) possibilities) `using` parList rseq
        wrappedPossibilities = map Just possibilities
        scorePossPairs = zip scores wrappedPossibilities in
    minOrMax scorePossPairs
```

V. Results and Conclusion

In order to test the results, two boards were tested with a depth limit that was kept constant across all functions in order to accurately compare the data. It is expected that the results will vary when tested with “harder” or more complicated boards. Tests were conducted with the starting mancala board defined as

```
let board = Board $ V.fromList [6,6,6,6,6,6,0,6,6,6,6,6,6,0]
```

and a fixed depth limit of 8.

	Avg. Running Time (s)
Minimax	4.249
AlphaBeta Pruning	0.764

Figure 3. Average running times of minimax and alphabeta pruning algorithms on starting board

The average running times of the sequential minimax and alphabeta pruning implementations are shown in Figure 3. The alpha beta pruning algorithm, even without parallelism, was 3.485 seconds faster than the standard minimax. This is equivalent to a 82.02% decrease in time.

Parallel Minimax						
Cores	1	2	4	6	8	10
Total Running Time (s)	4.456	2.438	1.344	1.916	2.173	2.073

Figure 4. Parallel Minimax running on increasing cores on starting board

Figure 4 displays the results of the parallel minimax algorithm using different amounts of cores. The largest difference in time is seen with 4 cores at 1.344 seconds which is 68.37% faster than the sequential minimax. There is an increase in the total time at 6, 8, and 10 cores which contradicts the idea that parallelism is used to increase performance. However, with the increase in new threads also comes an increase in the time needed to create all those threads. This starts to outweigh the benefits of parallelism especially since the algorithm was already running pretty quickly to begin with. This can also be the result of hardware limitations when it comes to how many threads the computer can make. With these results it can be inferred that alphabeta is the most optimized when it comes to the starting game board and an alphabeta parallel could possibly be even faster.

Parallel Minimax Threadscope and Runtime Data

```
(base) chanceonyiorah@Chances-MBP src % time ./Minimax +RTS -N1 -ls -s
(1,Just 0)
  12,805,823,000 bytes allocated in the heap
      54,384,616 bytes copied during GC
          83,440 bytes maximum residency (9 sample(s))
          29,056 bytes maximum slop
               3 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0     12319 colls,     0 par    0.160s   0.200s     0.0000s    0.0004s
  Gen  1         9 colls,     0 par    0.001s   0.001s     0.0002s    0.0003s

  TASKS: 4 (1 bound, 3 peak workers (3 total), using -N1)

  SPARKS: 1179392 (0 converted, 0 overflowed, 0 dud, 1132858 GC'd, 46534 fizzled)

  INIT    time    0.000s  (  0.008s elapsed)
  MUT     time    4.161s  (  4.236s elapsed)
  GC      time    0.162s  (  0.201s elapsed)
  EXIT    time    0.000s  (  0.011s elapsed)
  Total   time    4.323s  (  4.456s elapsed)

  Alloc rate    3,077,455,082 bytes per MUT second

  Productivity  96.2% of total user, 95.1% of total elapsed

./Minimax +RTS -N1 -ls -s  4.33s user 0.11s system 92% cpu 4.781 total
```

Figure 5. Performance data of parallel minimax on 1 core

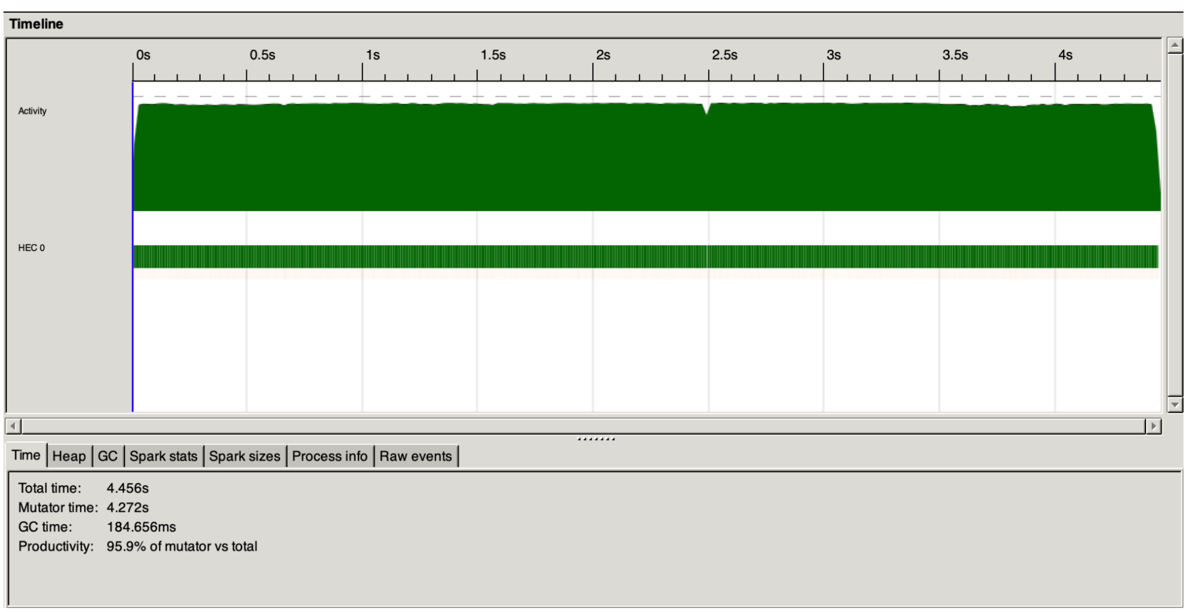


Figure 6. Parallel minimax on 1 core

```
(base) chanceonyiorah@Chances-MBP src % time ./Minimax +RTS -N2 -ls -s
(1,Just 0)
  12,806,112,784 bytes allocated in the heap
      52,965,488 bytes copied during GC
         168,144 bytes maximum residency (9 sample(s))
          36,256 bytes maximum slop
               4 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0      6282 colls,  6282 par    0.261s   0.162s     0.0000s    0.0009s
  Gen  1         9 colls,     8 par    0.002s   0.001s     0.0001s    0.0002s

  Parallel GC work balance: 74.01% (serial 0%, perfect 100%)

  TASKS: 6 (1 bound, 5 peak workers (5 total), using -N2)

  SPARKS: 1179392 (13 converted, 0 overflowed, 0 dud, 1132053 GC'd, 47326 fizzled)

  INIT    time    0.000s  (  0.007s elapsed)
  MUT     time    4.350s  (  2.265s elapsed)
  GC      time    0.263s  (  0.163s elapsed)
  EXIT    time    0.000s  (  0.002s elapsed)
  Total   time    4.614s  (  2.438s elapsed)

  Alloc rate    2,943,820,957 bytes per MUT second

  Productivity  94.3% of total user, 92.9% of total elapsed

./Minimax +RTS -N2 -ls -s  4.62s user 0.13s system 193% cpu 2.453 total
```

Figure 7. Performance data of parallel minimax on 2 cores

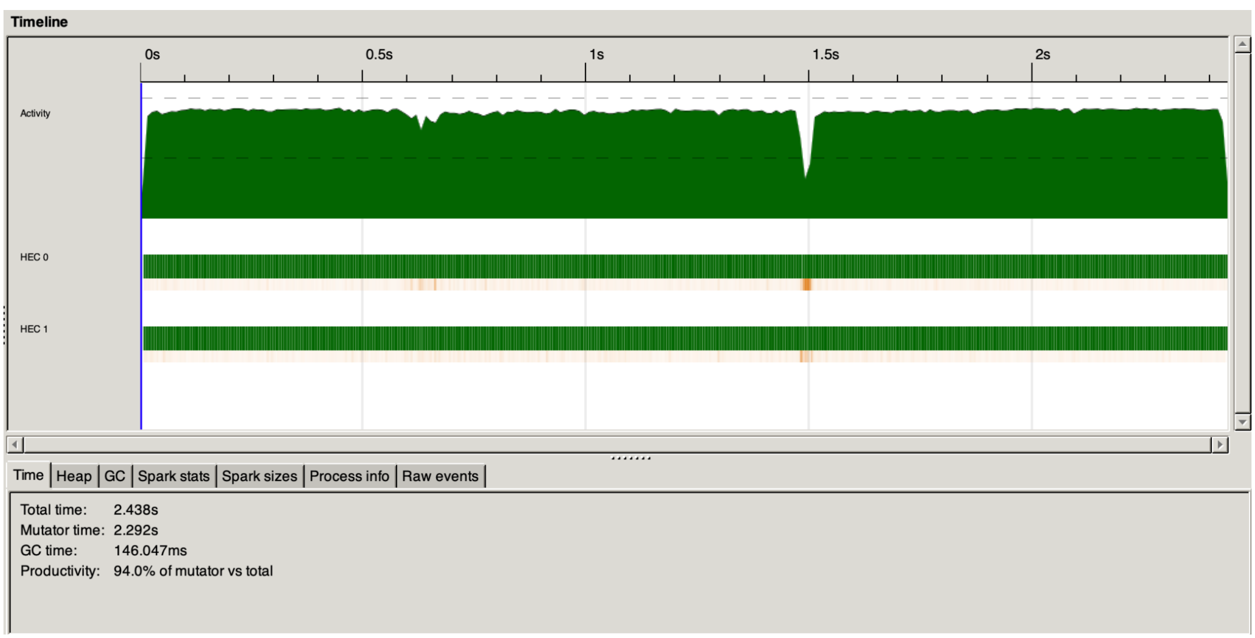


Figure 8. Parallel minimax on 2 cores

```
(base) chanceonyiorah@Chances-MBP src % time ./Minimax +RTS -N4 -ls -s
(1,Just 3)
  12,806,931,592 bytes allocated in the heap
      53,865,456 bytes copied during GC
         365,064 bytes maximum residency (12 sample(s))
          62,312 bytes maximum slop
               6 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0      3243 colls,  3243 par    0.418s   0.125s     0.0000s    0.0004s
  Gen  1        12 colls,    11 par    0.005s   0.002s     0.0002s    0.0006s

  Parallel GC work balance: 72.71% (serial 0%, perfect 100%)

  TASKS: 10 (1 bound, 9 peak workers (9 total), using -N4)

  SPARKS: 1179392 (70 converted, 0 overflowed, 0 dud, 1129435 GC'd, 49887 fizzled)

  INIT    time    0.001s  (  0.009s elapsed)
  MUT     time    4.395s  (  1.197s elapsed)
  GC      time    0.423s  (  0.127s elapsed)
  EXIT    time    0.000s  (  0.011s elapsed)
  Total   time    4.820s  (  1.344s elapsed)

  Alloc rate    2,913,658,730 bytes per MUT second

  Productivity  91.2% of total user, 89.1% of total elapsed

./Minimax +RTS -N4 -ls -s  4.82s user 0.13s system 299% cpu 1.654 total
```

Figure 9. Performance data of parallel minimax on 4 cores

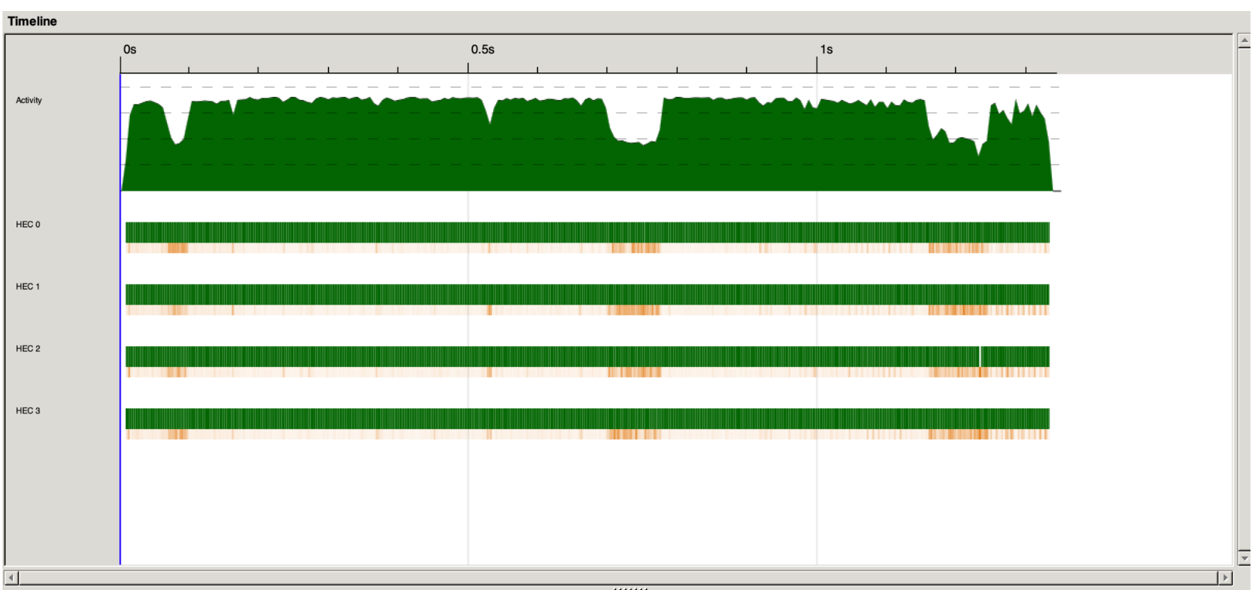


Figure 10. Parallel minimax on 4 cores

```
(base) chanceonyiorah@Chances-MBP src % time ./Minimax +RTS -N6 -ls -s
(1,Just 3)
  12,816,902,712 bytes allocated in the heap
      73,544,136 bytes copied during GC
         557,744 bytes maximum residency (32 sample(s))
          84,768 bytes maximum slop
               8 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0      3008 colls,  3008 par    1.816s   0.335s     0.0001s    0.0020s
  Gen  1        32 colls,    31 par    0.030s   0.010s     0.0003s    0.0047s

  Parallel GC work balance: 68.99% (serial 0%, perfect 100%)

  TASKS: 14 (1 bound, 13 peak workers (13 total), using -N6)

  SPARKS: 1179458 (1212 converted, 0 overflowed, 0 dud, 1098493 GC'd, 79753 fizzled)

  INIT    time    0.001s  (  0.009s elapsed)
  MUT     time    5.057s  (  1.556s elapsed)
  GC      time    1.846s  (  0.345s elapsed)
  EXIT    time    0.000s  (  0.006s elapsed)
  Total   time    6.904s  (  1.916s elapsed)

  Alloc rate    2,534,389,157 bytes per MUT second

  Productivity  73.2% of total user, 81.2% of total elapsed

./Minimax +RTS -N6 -ls -s  6.91s user 0.31s system 373% cpu 1.933 total
```

Figure 11. Performance data of parallel minimax on 6 cores

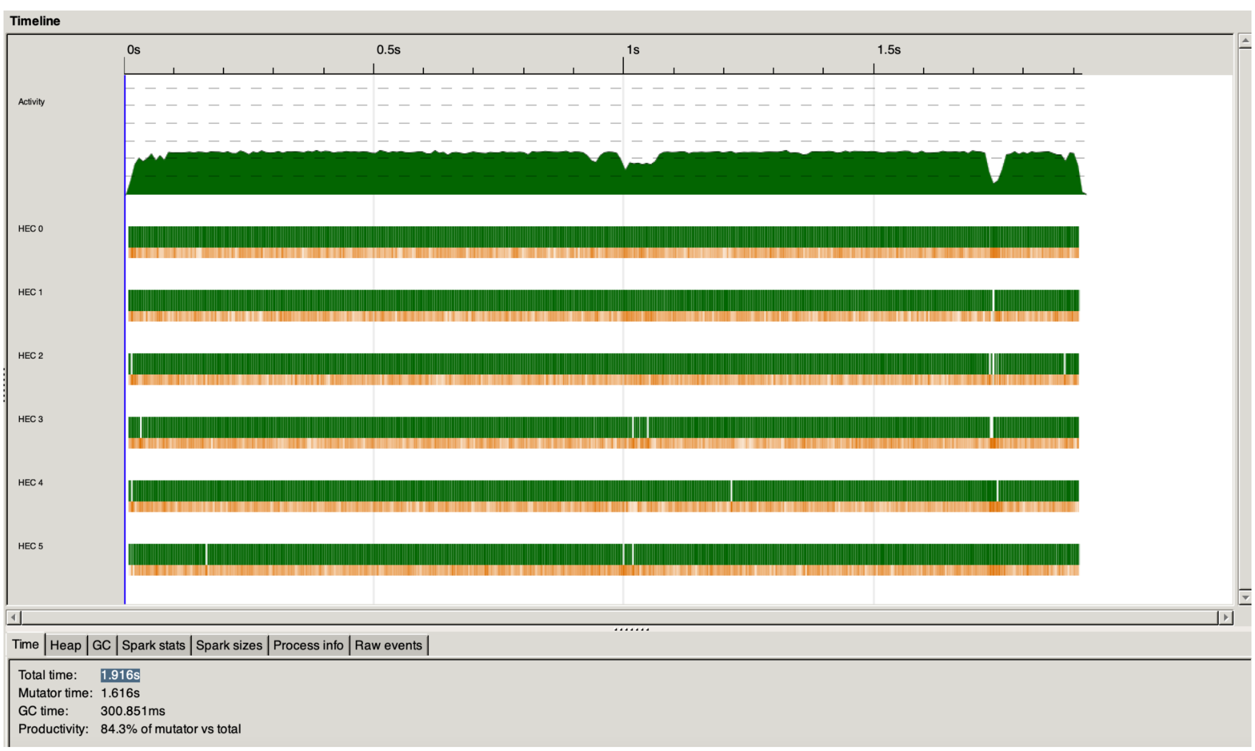


Figure 12. Parallel minimax on 6 cores

```
(base) chanceonyiorah@Chances-MBP src % time ./Minimax +RTS -N8 -ls -s
(1,Just 3)
  12,827,312,808 bytes allocated in the heap
      75,942,536 bytes copied during GC
         825,192 bytes maximum residency (48 sample(s))
         109,400 bytes maximum slop
              11 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0      2531 colls,  2531 par    1.828s   0.454s     0.0002s    0.0070s
  Gen  1        48 colls,    47 par    0.046s   0.012s     0.0002s    0.0005s

  Parallel GC work balance: 63.63% (serial 0%, perfect 100%)

  TASKS: 18 (1 bound, 17 peak workers (17 total), using -N8)

  SPARKS: 1179401 (2699 converted, 0 overflowed, 0 dud, 1078996 GC'd, 97706 fizzled)

  INIT    time    0.001s  (  0.008s elapsed)
  MUT     time    5.284s  (  1.693s elapsed)
  GC      time    1.874s  (  0.466s elapsed)
  EXIT    time    0.000s  (  0.006s elapsed)
  Total   time    7.159s  (  2.173s elapsed)

  Alloc rate    2,427,409,014 bytes per MUT second

  Productivity  73.8% of total user, 77.9% of total elapsed

./Minimax +RTS -N8 -ls -s  7.16s user 0.36s system 343% cpu 2.191 total
```

Figure 13. Performance data of parallel minimax on 8 cores

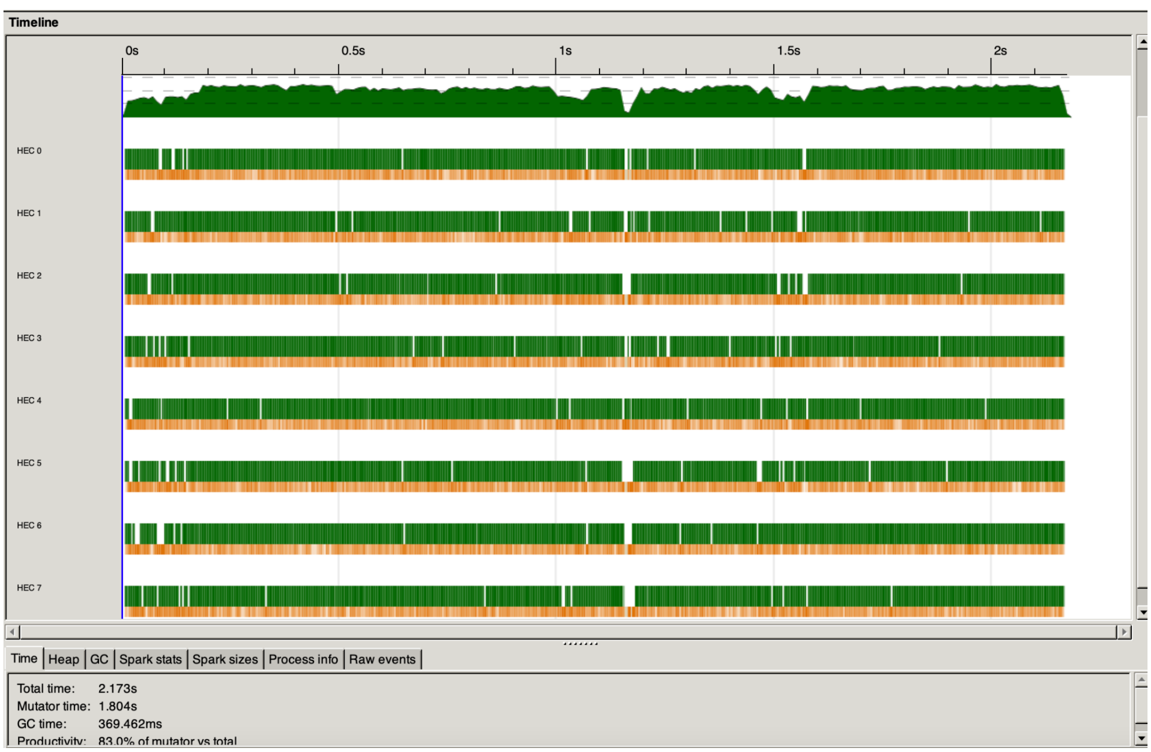
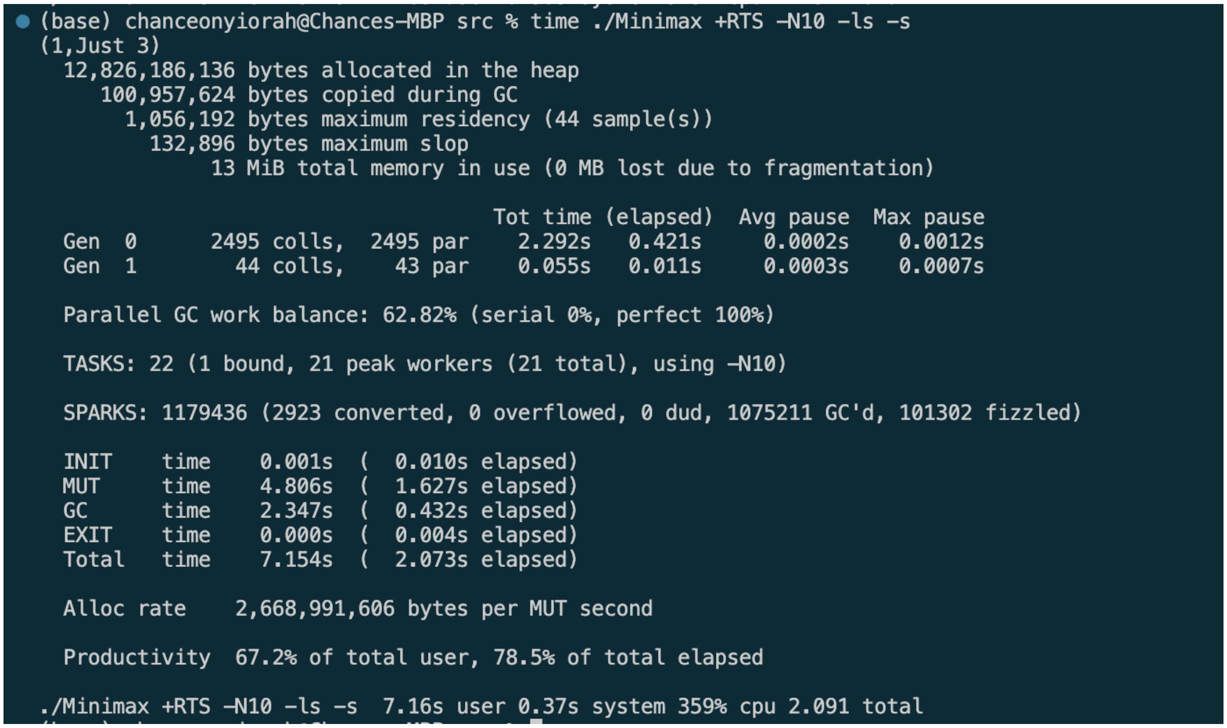


Figure 14. Parallel minimax on 8 cores



```
(base) chanceonyiorah@Chances-MBP src % time ./Minimax +RTS -N10 -ls -s
(1,Just 3)
  12,826,186,136 bytes allocated in the heap
     100,957,624 bytes copied during GC
       1,056,192 bytes maximum residency (44 sample(s))
         132,896 bytes maximum slop
              13 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0      2495 colls,  2495 par    2.292s   0.421s     0.0002s    0.0012s
  Gen  1        44 colls,    43 par    0.055s   0.011s     0.0003s    0.0007s

  Parallel GC work balance: 62.82% (serial 0%, perfect 100%)

  TASKS: 22 (1 bound, 21 peak workers (21 total), using -N10)

  SPARKS: 1179436 (2923 converted, 0 overflowed, 0 dud, 1075211 GC'd, 101302 fizzled)

  INIT    time    0.001s  (  0.010s elapsed)
  MUT     time    4.806s  (  1.627s elapsed)
  GC      time    2.347s  (  0.432s elapsed)
  EXIT    time    0.000s  (  0.004s elapsed)
  Total   time    7.154s  (  2.073s elapsed)

  Alloc rate    2,668,991,606 bytes per MUT second

  Productivity  67.2% of total user, 78.5% of total elapsed

./Minimax +RTS -N10 -ls -s  7.16s user 0.37s system 359% cpu 2.091 total
```

Figure 15. Performance data of parallel minimax on 10 cores

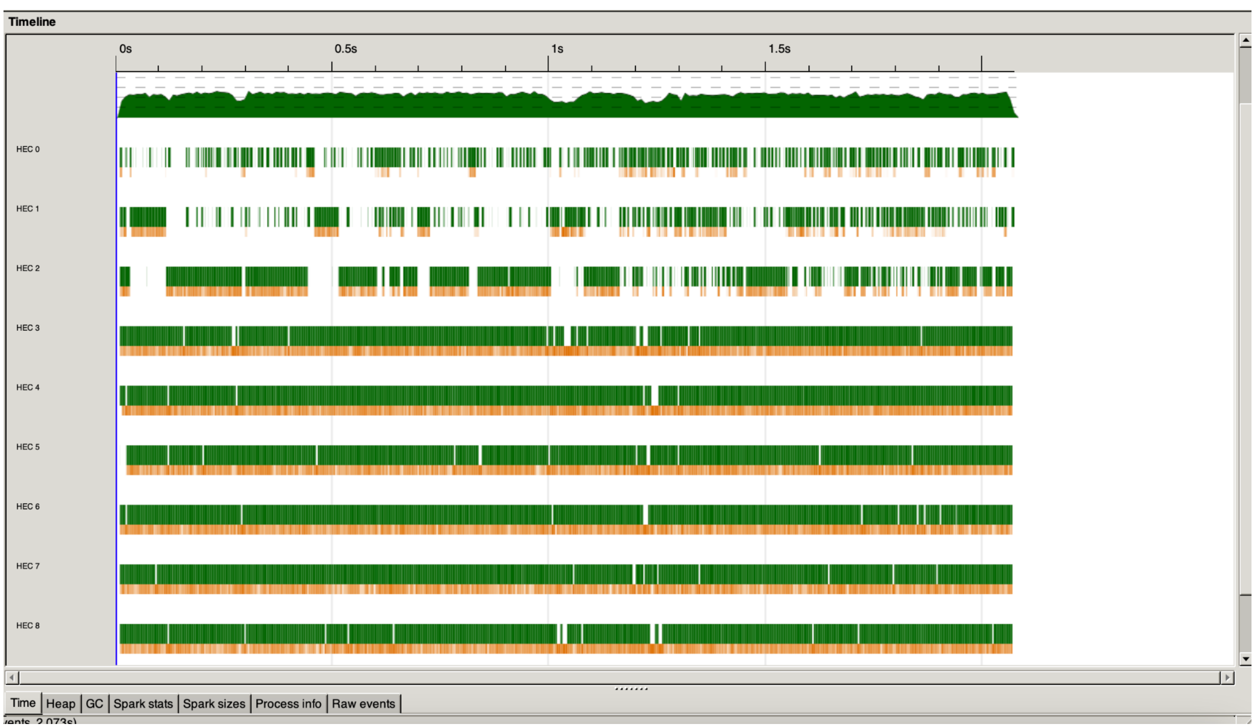


Figure 16. Parallel minimax on 10 cores

From these Threadscope graphs, we can see that the effects of parallelism on the load balancing between cores tapers off after 4 cores are used.

Board 2

```
let board = Board $ V.fromList [1,2,7,4,0,1,32,1,0,2,1,1,2,18]
```

	Avg. Running Time (s)
Minimax	3.707
AlphaBeta Pruning	1.823

Figure 17. Average running times of minimax and alphabeta pruning algorithms on Board 2

Parallel Minimax						
Cores	1	2	4	6	8	10
Total Running Time (s)	3.840	1.822	1.217	1.660	1.620	1.797

Figure 18. Parallel Minimax running on increasing cores on Board 2

In Figure 17 we can see that the alphabeta function again ran faster than the minimax algorithm, but with a significantly less speedup of 50.82%. Figure 18 also displays the results of the parallel minimax algorithm on Board 2 with the increasing number of cores. Again we see the least amount of time taken with 4 cores at 1.217 seconds which is 67.17% faster than the sequential minimax. In this case, the speedup was more consistent in comparison to the results with the starting board. This can lead us to think that the alphabeta function's optimization is dependent on the board and the efficiency of the original sequential minimax algorithm which already ran quickly to begin with.

VI. Code

Play.hs

```
module Main where
import           Data.List
import qualified Data.Vector                     as V
import           Mancala
import           Minimax
import           System.Exit
import           System.IO
import           Text.Printf

{-
Starting game board

   A  B  C  D  E  F
P   6  6  6  6  6  6
0                    0
   6  6  6  6  6  6  C
   L  K  J  I  H  G
-}

-- translate letters into numeric spaces
--valid computer moves
getComputerMove :: String -> Int
getComputerMove "L" = 0
getComputerMove "K" = 1
getComputerMove "J" = 2
getComputerMove "I" = 3
getComputerMove "H" = 4
getComputerMove "G" = 5
getComputerMove "q" = 13 --quit
getComputerMove "Q" = 13 --quit
getComputerMove _ = 14 --invalid input

--valid player moves
```

```haskell
getPlayerMove :: String -> Int
getPlayerMove "A" = 12
getPlayerMove "B" = 11
getPlayerMove "C" = 10
getPlayerMove "D" = 9
getPlayerMove "E" = 8
getPlayerMove "F" = 7
getPlayerMove "q" = 13 --quit
getPlayerMove "Q" = 13 --quit
getPlayerMove _ = 14 --invalid inpuy

-- translate numeric spaces into letters
getComputerLetter :: Int -> String
getComputerLetter 0 = "L"
getComputerLetter 1 = "K"
getComputerLetter 2 = "J"
getComputerLetter 3 = "I"
getComputerLetter 4 = "H"
getComputerLetter 5 = "G"
getComputerLetter _ = error "Invalid move"

--input move, check if move is valid
getMove :: Player -> Board -> IO Int
getMove p (Board b) = do
   str <- getLine
   let move = if p == Computer then getComputerMove str else getPlayerMove str
   if (move == 14 || ((b V.! move) == 0 && move /= 13)) --invalid letter or hole that
is empty
       then do
           putStr "Invalid Move. Try again: "
           hFlush stdout
           getMove p (Board b)
       else do
           if move == 13 --quit key
           then do
               putStrLn "Quitting."
               exitWith ExitSuccess
           else return move

--identify who's turn it is
printPlayer :: Player -> IO ()
printPlayer Computer = putStrLn "Computer: "
```

```haskell
printPlayer Player2 = putStrLn "You: "

--print marbles in each hole
printMarbles :: Board -> [Int] -> IO String
printMarbles (Board b) xs = do
   lineStr <-
       return
           (foldl (\str n -> str ++ (printf "%3d" n)) "" (map (\i -> b V.! i) xs)
           )
   return lineStr

--print hole letters on top along with marbles
printTopRow :: Board -> IO ()
printTopRow b = do
   str <- printMarbles b [12, 11 .. 7]
   putStrLn $ "        " ++ "   " ++ "A  B  C  D  E  F"
   putStrLn $ "        " ++ "P" ++ str

--print hole letters on bottom along with marbles
printBottomRow :: Board -> IO ()
printBottomRow b = do
   str <- printMarbles b [0 .. 5]
   putStrLn $ "        " ++ " " ++ str ++ "  C"
   putStrLn $ "        " ++ "   " ++ "L  K  J  I  H  G"

--print both stores
printStores :: Board -> IO ()
printStores (Board b) =
   putStrLn $ "        " ++ (show $ b V.! 13) ++ (replicate 20 ' ') ++ (show $ b V.!
6)

--print board
printBoard :: Board -> IO ()
printBoard b = do
   printTopRow b
   printStores b
   printBottomRow b

--print board and get game input
printGameState :: MancalaGameState -> IO ()
printGameState (MancalaGameState b p _) = do
   printPlayer p
```

```
   printBoard b

applyMove :: MancalaGameState -> Int -> IO ()
applyMove gs move = return (distributeMarbles gs move) >>= playGame

humanMoveGS :: MancalaGameState -> IO Int
humanMoveGS (MancalaGameState board player _) = do
   m <- getMove player board
   putStrLn ""
   return m

makeMoveGS :: MancalaGameState -> IO Int
makeMoveGS gs = do
   let (score, move) = minimaxPar gs False 0 8
   let Just x = move
   if move == Just x
       then do
           printf "Computer move: '%s'.\n\n"
                  (getComputerLetter x)
           return x
       else error "Invalid move: Nothing"

playGame :: MancalaGameState -> IO()
-- if game is over print results
playGame (MancalaGameState board computer player) | rowEmpty board Computer ||
rowEmpty board Player2 = do
   putStrLn $ "Game over. " ++ winString
   printGameState (MancalaGameState board computer player)
       where
           other   | player == Computer = Player2
                   | otherwise = Computer
           winString | (evaluate (MancalaGameState board computer player) > 0) =
"Winner is " ++ (show player)
                     | (evaluate (MancalaGameState board computer player) < 0) =
"Winner is " ++ (show other)
                     | otherwise = "Tie."
-- else print current game state and get next move
playGame (MancalaGameState board Computer Computer) = do
   printGameState (MancalaGameState board Computer Computer)
   putStrLn "Computer's turn"
   move <- makeMoveGS (MancalaGameState board Computer Computer)
   applyMove (MancalaGameState board Computer Computer) move
```

```haskell
playGame (MancalaGameState board Player2 x) = do
   printGameState (MancalaGameState board Player2 x)
   putStr "Enter move: "
   hFlush stdout
   move <- humanMoveGS (MancalaGameState board Player2 x)
   applyMove (MancalaGameState board Player2 x) move

startGameState = MancalaGameState initialBoard Computer Computer
main = do
   startGS <- return startGameState
   playGame startGS
```

Mancala.hs

```haskell
module Mancala where

import           Data.List
import qualified Data.Vector   as V
--import           Minimax

class GameState a where
   evaluate    :: a -> Int
   gameOver    :: a -> Bool
   possibleMoves    :: a -> [Int]
   makePossibility    :: a -> Int -> a
   isMaximizing    :: a -> Bool

data Player = Computer | Player2
   deriving (Eq, Show)

data Board = Board (V.Vector Int)
   deriving (Show)

initialBoard = Board $ V.fromList [6,6,6,6,6,6,0,6,6,6,6,6,6,0]

data MancalaGameState = MancalaGameState Board Player Player
   deriving Show
instance GameState MancalaGameState where
   evaluate (MancalaGameState board _ player) = getScore board player
   gameOver (MancalaGameState board _ _) = isGameOver board
   possibleMoves (MancalaGameState board player _) = getPossibleMoves board player
```

```haskell
   makePossibility = distributeMarbles
   isMaximizing (MancalaGameState _ computer player) = computer == player

rowEmpty :: Board -> Player -> Bool
rowEmpty (Board board) player =
   rowTotal == 0
   where rowTotal | player == Computer = board V.! 0 + board V.! 1 + board V.! 2 +
board V.! 3 + board V.! 4 + board V.! 5
                  | otherwise = board V.! 7 + board V.! 8 + board V.! 9 + board V.! 10
+ board V.! 11 + board V.! 12

mancalaTotal :: Board -> Player -> Int
mancalaTotal (Board board) player = board V.! (storePos player)

{-
Game is over when both rows are empty
Total 72 marbles are caught by players
-}
isGameOver :: Board -> Bool
isGameOver board
   | totalPoints == 72 = True
   | otherwise = False
   where totalPoints = mancalaTotal board Computer + mancalaTotal board Player2

storePos :: Player -> Int
storePos p | p == Computer = 6
           | otherwise = 13

getScore :: Board -> Player -> Int
getScore board player = (if (rowEmpty board Computer || rowEmpty board Player2) then
100 else 1) * (mancalaTotal board player - mancalaTotal board otherPlayer)
   where otherPlayer
           | player == Computer = Player2
           | otherwise = Computer

getPossibleMoves :: Board -> Player -> [Int]
getPossibleMoves (Board board) player =
   filter (\i -> (board V.! i) /= 0) rows
   where rows | player == Computer = [0..5]
              | otherwise = [7..12]

distributeMarbles :: MancalaGameState -> Int -> MancalaGameState
```

```
distributeMarbles (MancalaGameState (Board b) computer player) pos = (MancalaGameState
finalNewBoard nextPlayer player)
 where
   count           = (b V.! pos)
   boardGetMarbles = Board (b V.// [(pos, 0)])
   (newBoard, nextPlayer) = placeStones boardGetMarbles computer (pos + 1) count
   finalNewBoard | (rowEmpty newBoard Computer || rowEmpty newBoard Player2) =
endGameMove newBoard
                 | otherwise = newBoard

endGameMove :: Board -> Board
endGameMove (Board b) =
   Board
       $  b
       V.// (  [(6, (b V.! 6) + computerTotal), (13, (b V.! 13) + playerTotal)]
          ++ computerZeros
          ++ playerZeros
          )  where
   totalFunc = \l -> sum $ map (\i -> b V.! i) l
   computerTotal    = totalFunc [0 .. 5]
   playerTotal    = totalFunc [7 .. 12]
   zeroFunc   = map (\i -> (i, 0))
   computerZeros    = zeroFunc [0 .. 5]
   playerZeros    = zeroFunc [7 .. 12]

nextPos :: Player -> Int -> Int
nextPos player pos |  (player == Computer && pos == 12)  = 0
                   |  (player == Player2 && pos == 5)  = 7
                   | pos == 13 = 0
                   | otherwise = pos + 1

placeLastStone :: Board -> Player -> Int -> (Board, Player)
placeLastStone (Board board) player pos
   | board V.! pos == 0 && board V.! ((-) 12 pos) /= 0 && playerHole == player
   = (Board $ board V.// [(holeAcross, 0), (storePos player, newCount)], otherPlayer)
 where
   holeAcross       = (-) 12 pos
   holeAcrossCount = (board V.! holeAcross)
   newCount        = (mancalaTotal (Board board) player) + holeAcrossCount + 1
   otherPlayer     | player == Computer = Player2
                   | otherwise = Computer
   playerHole | pos >= 0 && pos <= 6  = Computer
```

```
              | otherwise = Player2
placeLastStone (Board board) player pos =
   (newBoard, nextPlayer)
   where newBoard = Board $ board V.// [(pos, (board V.! pos) + 1)]
         otherPlayer    | player == Computer = Player2
                        | otherwise = Computer
         nextPlayer     | pos == storePos player = player
                        | otherwise = otherPlayer

-- get only changes updates to update score
takeReverse :: [Int] -> Int -> [Int]
takeReverse listUpdates count = takeReverse' listUpdates count []
   where takeReverse' (x : _ ) 1     acc = x : acc
         takeReverse' (x : xs) count acc = takeReverse' xs (count - 1) (x : acc)

placeStones :: Board -> Player -> Int -> Int -> (Board, Player)
placeStones (Board b) player pos count = placeLastStone intermediateBrd player newPos
   where allUpdates  = iterate (\i -> nextPos player i) pos
         currUpdates = takeReverse allUpdates count
         intermediateBrd = Board $ b V.// (map (\l -> (head l, (b V.! head l) + length
l)) $ group . sort $ tail currUpdates)
         newPos = head currUpdates
```

Minimax.hs

```
module Minimax
where
import Debug.Trace
import Data.List
import Data.Ord
import Mancala
import qualified Data.Vector   as V
import Control.Parallel.Strategies

{-
--Test:
main :: IO()
main = do
   let board = Board $ V.fromList [6,6,6,6,6,6,0,6,6,6,6,6,6,0]
   let gs = MancalaGameState board Computer Player2
   --print (minimax gs False 0 8)
```

```
   --print (minimaxPar gs False 0 8)
   print (alphabeta gs 0 8 (-1000) 1000)
-}

-- ghc -threaded -rtsopts -eventlog --make -main-is Minimax  Minimax.hs -package
vector
-- time ./Minimax +RTS -ls -s
-- time ./Minimax +RTS -N2 -ls -s

minimax    :: (GameState a) => a -> Bool -> Int -> Int -> (Int, Maybe Int)
minimax gs _ depth depthlimit | depth == depthlimit || gameOver gs = (evaluate gs,
Nothing)
minimax gs minimize depth depthlimit =
   let minOrMax = (if minimize then minimumBy else maximumBy) (comparing fst)
       possibilities = (possibleMoves gs)
       scores = map fst $ map (\poss -> (minimax (makePossibility gs poss) (not
minimize) (depth+1) depthlimit)) possibilities
       wrappedPossibilities = map Just possibilities
       scorePossPairs = zip scores wrappedPossibilities in
   minOrMax scorePossPairs

minimaxPar    :: (GameState a) => a -> Bool -> Int -> Int -> (Int, Maybe Int)
minimaxPar gs _ depth depthlimit | depth == depthlimit || gameOver gs = (evaluate gs,
Nothing)
minimaxPar gs minimize depth depthlimit =
   let minOrMax = (if minimize then minimumBy else maximumBy) (comparing fst)
       possibilities = (possibleMoves gs)
       scores = (map fst $ map (\poss -> (minimaxPar (makePossibility gs poss) (not
minimize) (depth+1) depthlimit)) possibilities) `using` parList rseq
       wrappedPossibilities = map Just possibilities
       scorePossPairs = zip scores wrappedPossibilities in
   minOrMax scorePossPairs

{-
alphabeta   :: (GameState a) => a -> Int -> Int -> Int -> Int -> (Int, Maybe Int)
alphabeta gs _ _ _ _ | gameOver gs = (evaluate gs, Nothing)
alphabeta gs depth depthlimit _ _ | depth == depthlimit = (evaluate gs, Nothing)
alphabeta gs depth depthlimit alpha beta =
   alphabetafold possibilities alpha beta (-1)
   where   possibilities = possibleMoves gs
           alphabetafold [] a _ bestChild = (a, Just bestChild)
           alphabetafold (x:xs) a b bestChild =
```

```
               let child = makePossibility gs x
                   newAlpha = (if (isMaximizing child) then alphabetamax else
alphabetamin) child (depth+1) depthlimit a b in
               if (newAlpha >= b)
               then (newAlpha, Just x)
               else alphabetafold xs (max a newAlpha) b (if newAlpha > a then x else
bestChild)

alphabetamax    :: (GameState a) => a -> Int -> Int -> Int -> Int -> Int
alphabetamax gs _ _ _ _ | gameOver gs = evaluate gs
alphabetamax gs depth depthlimit _ _ | depth == depthlimit = evaluate gs
alphabetamax gs depth depthlimit alpha beta =
   alphabetaHelper gs possibilities alpha beta depth depthlimit
   where   possibilities = possibleMoves gs

alphabetamin    :: (GameState a) => a -> Int -> Int -> Int -> Int -> Int
alphabetamin gs _ _ _ _ | gameOver gs = evaluate gs
alphabetamin gs depth depthlimit _ _ | depth == depthlimit = evaluate gs
alphabetamin gs depth depthlimit alpha beta =
   alphabetaHelper gs possibilities alpha beta depth depthlimit
   where   possibilities = possibleMoves gs

alphabetaHelper :: (GameState a) => a -> [Int] -> Int -> Int -> Int -> Int -> Int
alphabetaHelper _ [] _ b _ _= b
alphabetaHelper gs (x:xs) a b depth depthlimit =
   let child = makePossibility gs x
       newBeta = (if (isMaximizing child) then alphabetamax else alphabetamin) child
(depth+1) depthlimit a b in
   if (newBeta <= a)
       then newBeta
       else alphabetaHelper gs xs a (min b newBeta) depth depthlimit
-}
```

References

[1] https://www.baeldung.com/java-minimax-algorithm

[2] https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/amp/

[3] https://www.cs.cornell.edu/courses/cs312/2002sp/lectures/rec21.htm

[4] https://www.oreilly.com/library/view/parallel-and-concurrent/9781449335939/ch03.html

[5]

https://medium.com/@aleksandrasays/brief-normal-forms-explanation-with-haskell-cd5dfa94a157#:~:text =An%20expression%20is%20in%20weak,would%20be%20in%20normal%20form.

[6] https://github.com/vagueanxiety/mancala (Referenced for code)

[7] https://gitlab.haskell.org/Abhiroop/nofib/-/blob/master/parallel/minimax/Game.hs (Referenced for code)