Beza Amsalu : baa2159
Miira Efrem : me2738
COMS 4995: Parallel Functional Programming
December 21, 2022

Connect4 Implemented with Parallelized Minimax Algorithm

1. Connect4 Background

Connect 4 is a game played by two agents, where each player has 21 balls, differentiated from each other by color or another characteristic. In our implementation, the player is “O” while the computer is “X”. The players take turns dropping a ball into a grid of six rows by seven columns, where the ball fills the lowest unoccupied slot in that column. A player wins the game, if they achieve four balls adjacent vertically, horizontally, or vertically first.

We utilized the Minimax algorithm, a backtracking algorithm, that looks ahead to pick a position that will either maximize the chance of the computer winning, or conversely, the move that will minimize the player’s ability to win. This algorithm is utilized to decide the best possible move for the computer to take. Because of the recursive nature of the minimizer and maximizer functions and the algorithm’s exponential runtime, we saw the implementation of this game as a good candidate for parallelization. The minimax algorithm was implemented using a search tree whose nodes held possible game states represented by its corresponding board.

The nature of the minimax algorithm forces it to traverse the entire tree for the best move, so we utilized the static pruning to minimize the number of nodes that have to be evaluated by the minimax algorithm, which greatly improves its performance, especially with parallelization.

2. Solution

2.1 Game representation

We defined a new data type, BoardEntry to represent the entry that occupies a slot in the game board, or indicates that it is empty.

```
data BoardEntry = O | E | X deriving (Ord, Show)
```

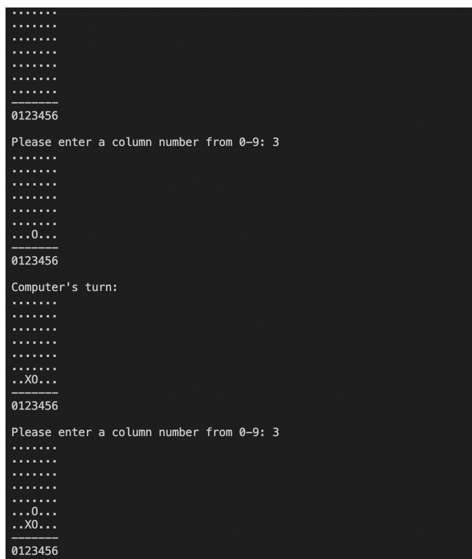
We utilize the Ord class for the minimizer and maximizer during the implementation of minimax, and the Show class to print and display the BoardEntries on the terminal during game play.

Therefore a column is represented by an array of BoardEntries, and the whole game board is an array of Column objects.

```
type Column = [BoardEntry]
type Board = [Column]
```

2.2 Implementation

The game board is represented in the terminal with a six by seven array of dots, initially when it is empty, and each dot is replaced with a BoardEntry, either “X” or “O” when a move is made in that position. When it is the human player’s turn, the terminal prompts them to input which column for their next move. It then verifies that their input is valid prior to making it. This checks if the input is a valid column number within the board, and verifies that both that column and the entire board are not full. Once verified, their entry is dropped to occupy the lowest available position in the column. Then the minimax algorithm is run to decide the computer’s move, and then an updated board is printed with its move. The players go back and forth until there is a winner or draw when the board is filled.



The game and its functions represent the moves of the two players, which update the Board object. The game search tree holds all possible formations of the board with the possible moves that each player can make. Our implementation generates a game tree and conducts static pruning to a specified depth. Each node is a possible game state represented by a board, and the leaves of the tree are boards in which there is a winner. Therefore each board produced is checked to see if there is a winner. If there is none, the winner corresponds to the E, or empty entry. The algorithm then works up the tree. If it is the player's turn, it takes the minimum of its immediate children, otherwise it's the computer which takes the maximum as its board's labeled "winner". Since the BoardEntry type implements Ord, the order of their "score" from least to greatest is O, E, X as listed in its initialization. Once the whole tree is generated, the best move for the root node results in the child with the same evaluation or "winner". If there are multiple children with the same evaluation, it selects a random one out of those children.[1]

```
bestMove :: BoardEntry -> [Column] -> Board
bestMove entry board = best_moves !! random
      where
        gametree = prune depth (generateTree board entry)
        Node (_, best) xss = ifMode mode entry gametree
        best_moves = [b' | Node (b', e') _ <- xss, e' == best]
        random  = unsafePerformIO (randomRIO (0,length best_moves
-1))
```

After each move, the game checks to see if a player has won. This is seen in the minimax algorithm when the board result of that move is a leaf node in the game search tree. It then checks all rows, columns , and diagonals for 4 consecutive BoardEntries. This case is seen in the first line of the minimax function, the sequential implementation of the minimax algorithm. If it is not a leaf, minimax is called recursively on the rest of the tree.

```
minmax ::  BoardEntry -> Tree Board -> Tree (Board, BoardEntry)
minmax _ (Node b []) = Node (b, findWinner b) []
minmax entry (Node b xss) = Node (b, evaluate evals) xss'
        where
          xss' = map (minmax (nextEntry entry)) xss
          evals = [e' | Node (_,e') _ <- xss']
          evaluate = if isMaximizing entry then maximum else minimum
```

So on the computer's turn, its best move is found by simulating the minimax algorithm on the game tree. If there is no move that directly results in a win, it randomly selects one of

[1] https://www.youtube.com/watch?v=IoCINPnt0us

the children of the root node, the current game state, that would result in the same evaluation.

2.3 Parallelization

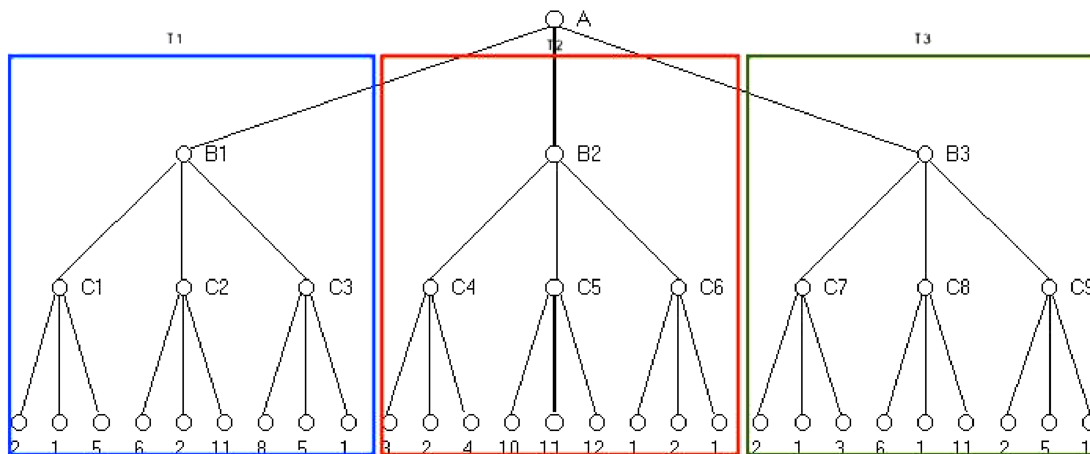
To enhance the efficiency of our minimax algorithm, we implemented two approaches: minimaxPar and minimaxParTwo. These approaches use parallel processing to run the sequential minimax algorithm for each child of the root node in the game tree.

```
minmaxPar :: BoardEntry -> Tree Board -> Tree (Board, BoardEntry)
minmaxPar _ (Node b []) = Node (b, findWinner b) []
minmaxPar entry (Node b xss) = Node (b, evaluate evals) xss'
        where
          xss' = map (minmax (nextEntry entry)) xss `using` parList
rdeepseq
          evals = [e' | Node (_,e') _ <- xss']
          evaluate = if isMaximizing entry then maximum else minimum
```

The former uses `rdeepseq` to fully evaluate each node before moving on to the next one, while minimaxParTwo utilizes `parMap` to evaluate the nodes, corresponding to each possible child Board, in parallel.

```
minmaxParTwo :: BoardEntry -> Tree Board -> Tree (Board, BoardEntry)
minmaxParTwo _ (Node b []) = runPar $ return $ Node (b, findWinner b)
[]
minmaxParTwo entry (Node b xss) = runPar $ do
        xss' <- parMap (minmax (nextEntry entry)) xss
        let evals = [p | Node (_, p) _ <- xss']
        let evaluate = if isMaximizing entry then maximum else
minimum
        return $ Node (b, evaluate evals) xss'
```

Our initial approach involved parallelizing the evaluation of every node in the game tree, but the granularity of these sparks was too high and, as a result, any performance gain from parallelization was dwarfed by the overhead of managing the sheer number of sparks. To mitigate this, we modified our approach to only run the minimax algorithm in dynamic chunks that are equal to the number of possible moves from the root node. This allows us to better utilize parallel processing and improve the overall speed of the algorithm.



partitioning game tree at depth to exploit parallelism

Our initial attempt at using parList with the `rseq` strategy did not provide the performance improvements we were seeking, so we implemented the `rdeepseq` strategy instead. To use this strategy, we had to write an instance of *NFData* for the BoardEntry data type that is evaluated in the game tree, while our game tree inherited this instance as it was implemented in Data.Tree. This approach yielded significant performance improvements when compared to the sequential version of the minimax algorithm. To measure the performance of the different implementations (sequential, minimaxPar, and minimaxParTwo), we measured the time it takes the minimax algorithm to generate the best move on an empty game tree, since this represents the worst-case running time for the minimax algorithm. The elapsed time for a full simulated game is roughly equal to the sum of the runtime for the computer to generate best moves until the game is finished. The results for the first parallelization attempt are shown below.

```
35,224,638,432 bytes allocated in the heap
   270,481,840 bytes copied during GC
    35,766,816 bytes maximum residency (12 sample(s))
       507,360 bytes maximum slop
            94 MiB total memory in use (0 MB lost due to fragmentation)

                                   Tot time (elapsed)  Avg pause  Max pause
Gen  0      6065 colls,  6065 par    5.265s   0.473s     0.0001s    0.0003s
Gen  1        12 colls,    11 par    0.218s   0.039s     0.0033s    0.0120s

Parallel GC work balance: 65.77% (serial 0%, perfect 100%)

TASKS: 26 (1 bound, 25 peak workers (25 total), using -N12)

SPARKS: 7 (6 converted, 0 overflowed, 0 dud, 0 GC'd, 1 fizzled)

INIT    time    0.001s  (  0.004s elapsed)
MUT     time   13.636s  (  2.006s elapsed)
GC      time    5.483s  (  0.512s elapsed)
EXIT    time    0.001s  (  0.008s elapsed)
Total   time   19.121s  (  2.529s elapsed)

Alloc rate    2,583,195,397 bytes per MUT second
```

Parallel minimax algorithm (one) generating the best move for worst case on a game tree of depth 6

Our second attempt to parallelize was using the `parMap` function and the underlying logic is essentially the same, in that we wanted to run the minimax algorithm on a dynamic chunk size equal to the number of possibleMoves from a given board. To this end, we used `parMap`, which spawns a computation and waits for its completion in parallel in conjunction with runPar which takes the return value of the parMap function out of the *Par* Monad. This attempt to parallelize the minimax algorithm yielded marginally better results compared to the first parallel algorithm, however, its underlying implementation of the parallelization was unexpected in that it did not create any sparks, while we expected it to create the same amount of sparks as the first method of parallelization, which is equal to the number of times of the child nodes of the root board (7) times however many times the game had to simulate best move for the computer before the game terminated.

```
35,224,710,024 bytes allocated in the heap
   273,319,944 bytes copied during GC
    38,355,632 bytes maximum residency (12 sample(s))
       560,464 bytes maximum slop
            99 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
Gen  0      5752 colls,  5752 par    3.978s   0.318s     0.0001s    0.0007s
Gen  1        12 colls,    11 par    0.233s   0.044s     0.0037s    0.0138s

Parallel GC work balance: 65.56% (serial 0%, perfect 100%)

TASKS: 26 (1 bound, 25 peak workers (25 total), using -N12)

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

INIT    time    0.001s  (  0.004s elapsed)
MUT     time   11.555s  (  1.768s elapsed)
GC      time    4.210s  (  0.363s elapsed)
EXIT    time    0.001s  (  0.006s elapsed)
Total   time   15.767s  (  2.140s elapsed)

Alloc rate    3,048,551,164 bytes per MUT second

Productivity  73.3% of total user, 82.6% of total elapsed
```

Moreover, there is room for further optimizing performance of the algorithm by building lazy minimum and maximum functions, which would stop looking for minimum/maximum in a list when it finds the first minimum/maximum. This function would yield similar results to a minimax function with alpha-beta pruning. As it is very similar in nature other than the fact that it evaluates all of the trees before pruning them , where as alpha-beta pruning wouldn't even consider the nodes in the first place.,

Performance Analysis

Note: To measure speed improvements we ran the minimax algorithm once on different depths, because the AI vs AI and AI vs Human game simulations don't always end with the same number of moves. Also, the total run time for a full simulation is the number of moves * time elapsed for one run of the minimax function thus it is an accurate heuristic to measure the speed of the program.

At depth 6:

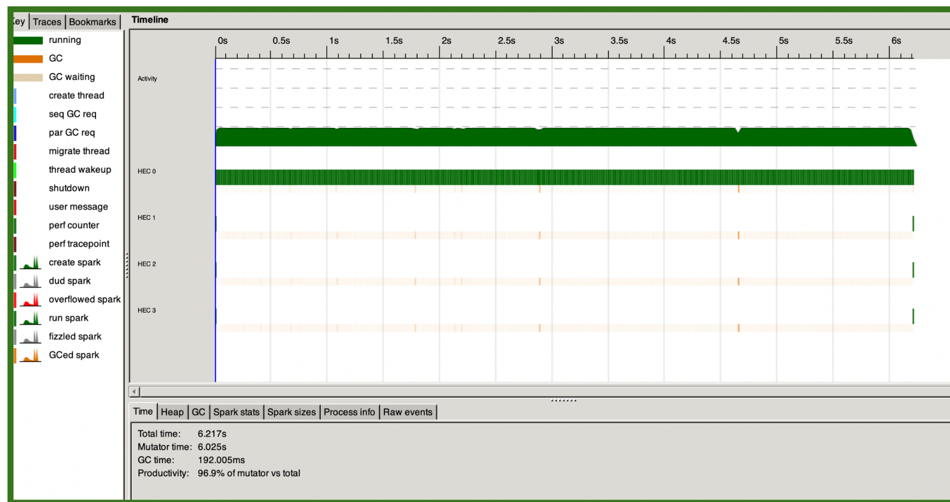
	Sequential	Parallel (s.1)	Parallel (s.2)
Total Time Elapsed (sec)	10.208	2.529	2.14
Speed Up (x)	-	4.04x	4.77x

At depth 7:

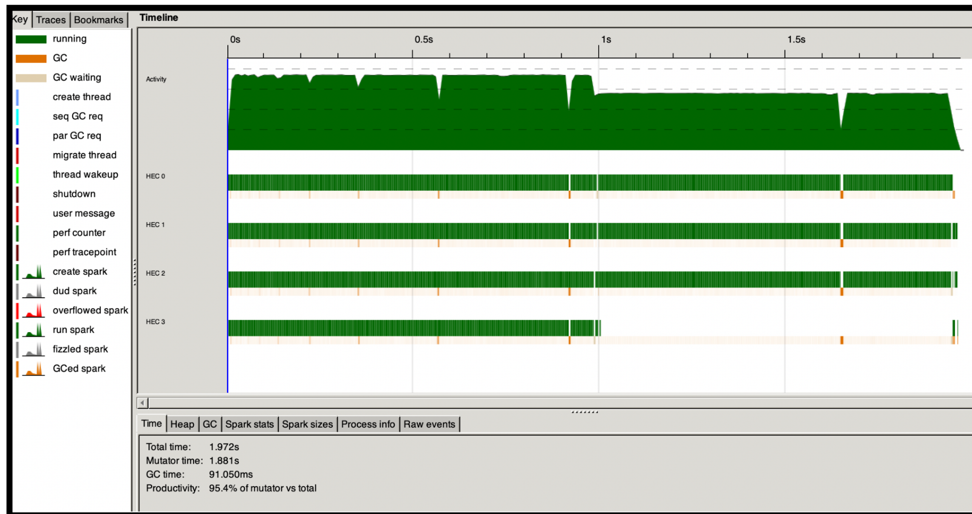
	Sequential	Parallel (s.1)	Parallel (s.2)
Total Time Elapsed (sec)	104.873	33.271	26.961
Speed Up (x)	-	3.152	3.890

Depth 6

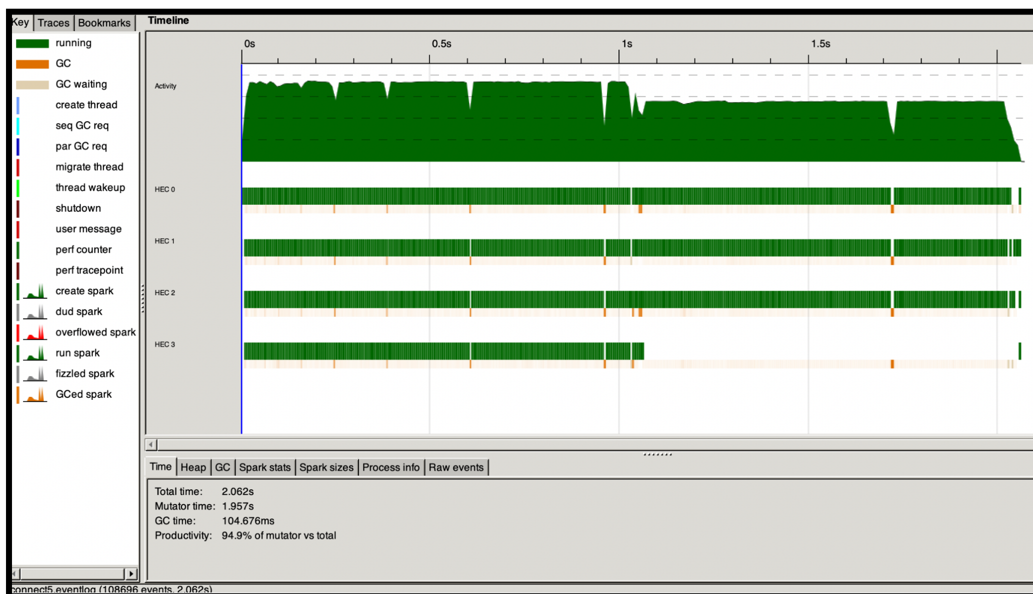
1. **Seq**



2. **MinmaxPar**



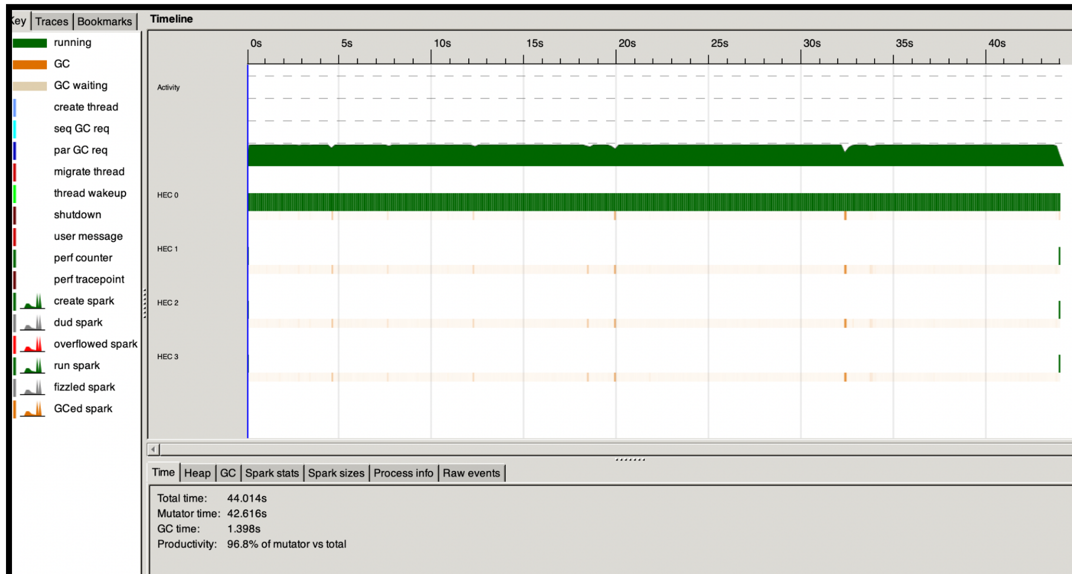
3. **MinmaxParTwo**



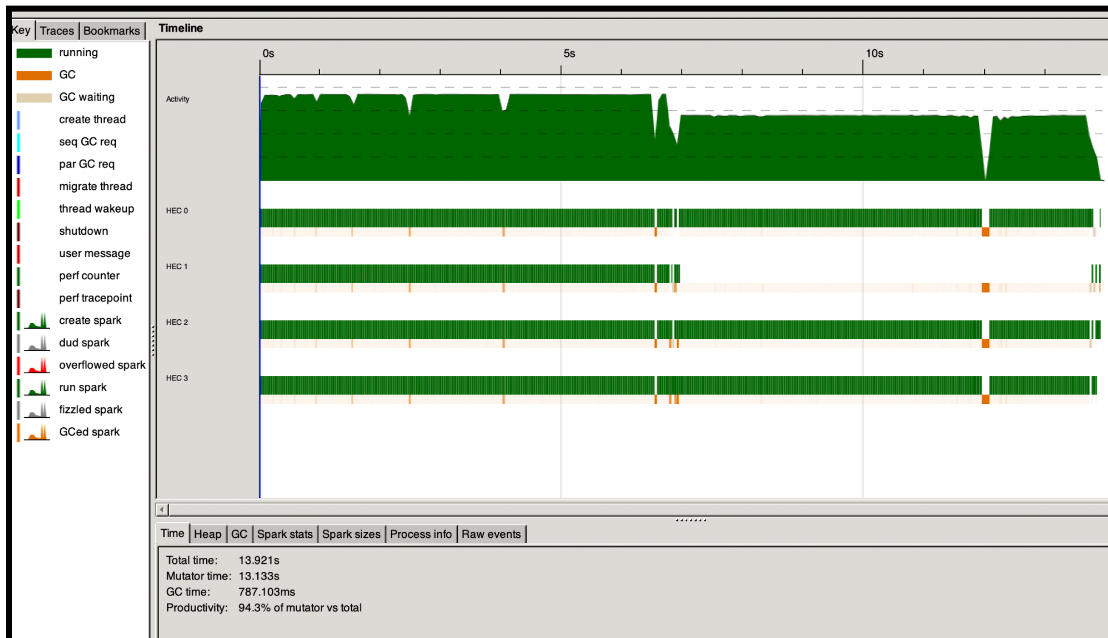
With 4-core processing and a tree depth of 6, the time taken for the computer to make its best move goes from 6.217 s in the sequential implementation to 1.972s with `minimaxPar` and 2.062s with `minimaxParTwo`. The productivity between the three different implementations does not seem to differ by a significant amount. The chart shows that for both parallel implementations, overall activity drops after 1s, as activity ceases in both of the fourth cores. With much smaller dips before this point, we see a significant drop at about 1.7s for garbage collection.

Depth 7

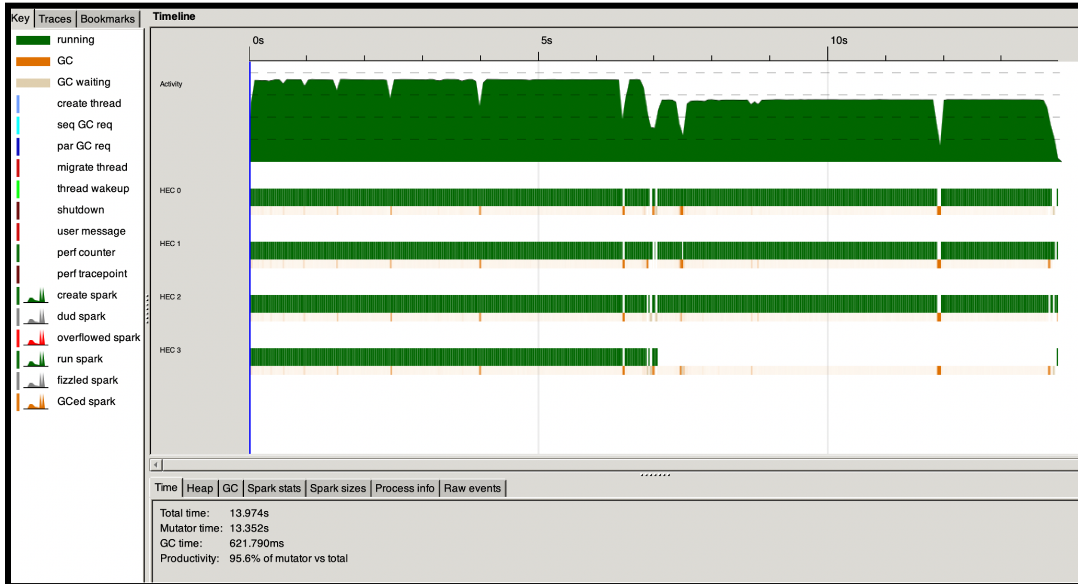
1. **Seq**



2. **MinmaxPar**



3. MinmaxParTwo



With depth 7, still using 4-core processing, the time taken for the two parallel minimax implementations take just about the same time, 13.921s and 13.974s for minmaxPar and minmaxParTwo, respectively, both still significantly faster than the sequential's 44.01s. Similar to depth 6, activity takes a dip, here around 7s for both parallel implementations, where activity ceases in the second core for `minmaxPar` and the fourth for `minmaxParTwo`.

3. Code Listing

— Connect4.hs –

```
{-# LANGUAGE InstanceSigs #-}
{-# OPTIONS_GHC -Wno-incomplete-patterns #-}

module Connect4 (run) where

import Prelude
import Data.List (transpose, tails)
import Data.Bool ()
import Data.Char (isDigit)
import System.Random ( randomRIO ) -- cabal install --lib random
import System.IO.Unsafe ( unsafePerformIO )
import Control.DeepSeq (NFData (rnf))
import Control.Parallel.Strategies
    ( parList, rdeepseq, using )
import Control.Monad.Par ( runPar, parMap ) -- cabal install --lib monad-par
import Data.Tree

data BoardEntry = O | E | X deriving (Ord, Show)
type Column = [BoardEntry]
type Board = [Column]

instance NFData BoardEntry where
  rnf x = x `seq` ()

instance Eq BoardEntry where
    (==) :: BoardEntry -> BoardEntry -> Bool
    O == O = True
    X == X = True
    E == E = True
    _ == _ = False


mode :: [Char]
mode = "par2"

ifMode :: [Char] -> BoardEntry -> Tree Board -> Tree (Board, BoardEntry)
ifMode "seq" = minmax
ifMode "par1" = minmaxPar
ifMode "par2" = minmaxParTwo

showPlayer :: BoardEntry -> Char
showPlayer O = 'O'
showPlayer E = '.'
showPlayer X = 'X'

-- Returns a column in the board given at a specific index
getCol :: Board -> Int -> [BoardEntry]
getCol board index = board !! index

gridSize :: Int
gridSize = 7
```

```
depth :: Int
depth = 6

-- Number of consecutive equivalent entries needed to meet winning condition
win :: Int
win = 4

-- Game mechanics --

-- Initializes an empty board
initEmptyBoard :: Board
initEmptyBoard = replicate gridSize $ replicate gridSize E

-- Drops an entry into the lowest unoccupied position of a column
dropEntry :: [BoardEntry] -> BoardEntry -> [BoardEntry]
dropEntry column entry =
            let filled_entries = filter (/=E) column
                empty_entries =  gridSize - length filled_entries
            in  if empty_entries == 0 then
                 column
                else
                replicate (empty_entries - 1) E ++ [entry] ++ filled_entries

-- Adds an new entry into the board and updates the board
makeMove :: Int -> BoardEntry -> Board -> Board
makeMove i entry board =  take i board ++ [dropEntry (getCol board i) entry] ++ drop (i + 1)
board

-- Checks if the move the player wants to make is valid
isValid :: Board -> Int -> Bool
isValid board index
                    | index >= gridSize || 0 > index = False
                    | head (board !! index) /= E = False
                    | otherwise = True

diagonals :: Board -> Board
diagonals []       = repeat []
diagonals (xs:xss) = takeWhile (not . null) $
    zipWith (++) (map (:[]) xs ++ repeat [])
                 ([]:diagonals xss)

-- Finds the diagonals and anti-diagonals of the original grid representation
allDiagonals :: Board -> [Column]
allDiagonals xss = diagonals (transpose xss) ++ diagonals (rotate90 xss)
    where rotate90 = reverse


-- Checks if a specific player has won
hasWon :: Board -> BoardEntry -> Bool
hasWon board entry = any (all (==entry)) (rows ++ cols ++ diags)
  where
    rows = winChunks board
    cols = winChunks (transpose board)
    diags = winChunks (allDiagonals board)

-- returns all sub rows for each row in the grid representation of the board
winChunks :: Board -> [Column]
```

```
winChunks = concatMap slidingWindow

-- Returns every subset of consecutive elements in a list the size of the winning condition
slidingWindow :: Column -> [Column]
slidingWindow xs = [take win xs' | xs' <- tails xs, length xs' >= win]

-- Generate a list of nextEntry moves
possibleMoves :: Board -> BoardEntry -> [Board]
possibleMoves board entry | isGameOver board = []
                          | otherwise = [makeMove i entry board | i <- [0..(gridSize-1)] ,
isValid board i]

-- Check if board is full
isFull :: Board -> Bool
isFull = notElem E . concat

-- Check if board is full and there are no empty spaces
isDraw :: Board -> Bool
isDraw board = isFull board && findWinner board == E

-- Outputs the nextplayer given the last player
nextEntry :: BoardEntry -> BoardEntry
nextEntry O = X
nextEntry X = O
nextEntry E = E

isGameOver :: Board -> Bool
isGameOver board = findWinner board /= E || isDraw board

--If player has won return X
--If computer has won return O
--If draw or inconclusive game return E
findWinner :: Board -> BoardEntry
findWinner board | hasWon board O = O
                 | hasWon board X = X
                 | otherwise = E



-- Game logic (AI) ---

-- Generates a gameSearchTree and prunes to a specified depth
-- If a winning board is reached, add board as a leaf in the game tree
generateTree :: Board -> BoardEntry -> Tree Board
generateTree board entry = Node board [generateTree b (nextEntry entry) | b <- possibleMoves
board entry]

-- static depth pruning for gametree
prune :: Int -> Tree Board -> Tree Board
prune 0 (Node x _)  = Node x []
prune n (Node x ts) = Node x [prune (n-1) t | t <- ts]

-- X is maximizing
-- O is minimizing
isMaximizing :: BoardEntry -> Bool
isMaximizing entry
            | entry == O = False
            | entry == X = True
```

```
--naive minimax algorithm
--labels leaf nodes with winner
--propagates evaluation up the game tree
--If computer is playing take minimum of the children
--If player is playing take the maximum of the children
--output tree
minmax ::  BoardEntry -> Tree Board -> Tree (Board, BoardEntry)
minmax _ (Node b []) = Node (b, findWinner b) []
minmax entry (Node b xss) = Node (b, evaluate evals) xss'
        where
          xss' = map (minmax (nextEntry entry)) xss
          evals = [e' | Node (_,e') _ <- xss']
          evaluate = if isMaximizing entry then maximum else minimum

-- Parallelizes by using a strategy (parList rdeepseq) to evaluate the tree in chunks equal to
the
-- Number of possible moves from the root tree
minmaxPar :: BoardEntry -> Tree Board -> Tree (Board, BoardEntry)
minmaxPar _ (Node b []) = Node (b, findWinner b) []
minmaxPar entry (Node b xss) = Node (b, evaluate evals) xss'
        where
          xss' = map (minmax (nextEntry entry)) xss `using` parList rdeepseq
          evals = [e' | Node (_,e') _ <- xss']
          evaluate = if isMaximizing entry then maximum else minimum


-- Parallelism by applying parMap to evaluate the tree in chunks equal to the
-- Number of possible moves from the root tree
minmaxParTwo :: BoardEntry -> Tree Board -> Tree (Board, BoardEntry)
minmaxParTwo _ (Node b []) = runPar $ return $ Node (b, findWinner b) []
minmaxParTwo entry (Node b xss) = runPar $ do
        xss' <- parMap (minmax (nextEntry entry)) xss
        let evals = [p | Node (_, p) _ <- xss']
        let evaluate = if isMaximizing entry then maximum else minimum
        return $ Node (b, evaluate evals) xss'

-- Given current board and the computer's entry, generate a gametree
-- Simulate minimax algorithm on the game tree
-- Randomly  pick one of the direct children of the root  node with the save evaluation
bestMove :: BoardEntry -> [Column] -> Board
bestMove entry board = best_moves !! random
      where
        gametree = prune depth (generateTree board entry)
        Node (_, best) xss = ifMode mode entry gametree
        best_moves = [b' | Node (b', e') _ <- xss, e' == best]
        random  = unsafePerformIO (randomRIO (0,length best_moves -1))


-- Utility Functions --

-- given a gametree labeled with
-- prints the grid to the terminal
printBoard :: Board -> IO ()
printBoard board = printBoard' $ transpose board
  where  printBoard' b = putStrLn (unlines (map showRow b ++ [line] ++ [nums]))
              where
                showRow = map showPlayer
```

```
                line = replicate gridSize '-'
                nums = take gridSize ['0'..]

run :: IO()
run = gameLoop initEmptyBoard X

gameLoop :: Board -> BoardEntry -> IO()
gameLoop board entry = do
                   printBoard board
                   let prompt = "Please enter a column number from 0-9: "
                   userChoice <- getUserChoice board prompt
                   let pBoard = makeMove userChoice entry board
                   printBoard pBoard

                   if isGameOver pBoard
                    then
                      putStrLn "Player has won!"
                    else
                      do
                        let cBoard = bestMove (nextEntry entry) pBoard
                        if isGameOver cBoard
                          then do
                            printBoard cBoard
                            putStrLn "Computer has won!"
                          else do
                            gameLoop cBoard entry

simulate :: Board -> BoardEntry -> IO()
simulate board entry = do
                   printBoard board
                   let cBoard_one = bestMove entry board
                   printBoard cBoard_one

                   if isGameOver cBoard_one
                    then
                      putStrLn "Computer 1 has won!"
                    else
                      do
                        let cBoard = bestMove (nextEntry entry) cBoard_one
                        if isGameOver cBoard
                          then do
                            printBoard cBoard
                            putStrLn "Computer 2 has won!"
                          else do
                            simulate cBoard entry

getUserChoice :: Board -> String -> IO Int
getUserChoice board prompt = do
                   putStr prompt
                   xs <- getLine
                   if xs /= [] && all isDigit xs && isValid board (read xs) then
                      return (read xs)
                   else
                      do putStrLn "ERROR: Invalid number"
                         getUserChoice board prompt


--start program when executed
```

```
main :: IO()
main = run
```

Sources:

https://www.youtube.com/watch?v=IoCINPnt0us
https://simonmar.github.io/bib/papers/strategies.pdf