MonteCarlo-C4: A Monte Carlo evaluator for Connect-4

Grayson Newell (gln2109)

December 2025

1 Introduction

1.1 Overview

Monte Carlo game playing methods are well suited for parallelization as they rely on many random simulations to create a probability space of results. In this project, we implement a Monte Carlo evaluator for Connect-4 in Haskell and analyze how parallelization affects its performance.

1.2 Hardware

Brand: Apple
Model: M1 Pro
Cores: 8
Threads: 8

2 Sequential Implementation

2.1 Connect-4 Board

Our Connect-4 board representation is simply a nested array of Cells, which have type Maybe Player. Player data can be either Red or Yellow and is implemented as NFData. For working with these data structures, we implemented the following functions:

- **emptyBoard** - Returns an empty board with hardcoded size (6 rows, 7 columns).
- **availableMoves** - Returns an array of available columns given a board.
- **applyMove** - Returns a new board with a move applied given a board, a player, and a column.
- **applyMoveList** - Returns a new board with a list of moves applied given a board, a starting player, and an array of columns.
- **boardString** - Returns a string representation of a board.
- **checkWin** - Returns a winning Player if one exists otherwise Nothing given a board.
- **simulate** - Runs a random simulation and returns the winner if one exists given a board, starting player, and an StdGen.

These are the datatypes and functions on which our Monte Carlo evaluator operates, as described below.

```
. . . . . . .    . . . . . . .    . . . . . . .    . . . . . . .    . . . . . . .
. . . . . . .    . . . . . . .    . . . . . . .    . . . . . . .    . . . . . . .
. . . . . . .    . . . . . . .    . . . . . . .    . . . . R . .    . . . . R . .
. . . . . . .    . . . . . . .    . . . . Y . .    . . . . Y . .    . . . . Y . .
. . . Y Y . .    . . R . Y . .    . . . R Y . .    Y R . R Y . .    Y R Y R Y . .
. R R R Y . .    Y R R R Y . .    Y R R R Y . .    Y R R R Y . .    Y R R R Y . .
Player: Red      Player: Yellow   Player: Red      Player: Yellow   Player: Red
```

Figure 1: Test Boards

2.2 Monte Carlo Evaluator

The Monte Carlo evaluator returns a predicted best move given a board and a player. Our implementation works as follows:

1. Take all available moves for the given board and create new boards with each move applied.
2. From each new board, run a specified number of simulations where each player makes random moves until the game ends.
3. Count how many simulations resulted in a win for the given player for each available move.
4. Return the move with the most simulated wins.

This is a very simple Monte Carlo implementation, as simulations are played randomly with no heuristics or tree structure. However for a game with a relatively low branching factor like Connect-4, predicting "good" moves is still possible if running enough simulations. We found 4096 simulations to be a sufficient value for strategic play and to emphasize parallel speedup.

2.3 Performance Testing

Figure 1 shows a set of five 7x6 Connect-4 boards that will be used for testing. These were chosen for their balance of progression and win condition.

We tested the performance of our evaluator by measuring the time taken to return an optimal move for all five of these boards. Over five trials, our sequential implementation took an average of 5.787 seconds.

Given our usage of deterministic pseudo-random number generation, the results for each trial were the same: [0, 4, 4, 3, 1]. These numbers represent the most optimal predicted column for each test board and player. Here, its apparent that the evaluator is capable of making educated moves, as it both obtains and blocks a win in the first and third samples respectively.

3 Parallelization

3.1 Initial Approach

Our parallel approach utilized parMap and rdeepseq from Control.Parallel.Strategies. Using these functions, we experimented with evaluating both available moves and game simulations in parallel. The results are shown below for the aforementioned sample boards, running 4096 simulations at 8 threads, averaged over five trials.

Parallelization	Average Runtime (s)
Available Moves	1.182
Game Simulations	0.994
Both	0.984

This data clearly demonstrates that parallelization of game simulations is more effective than doing so for available moves in terms of performance. So, we chose to parallelize game simulations in our final implementation. While the runtimes for using nested parallelization may have been marginally better, this implementation raises scalability concerns and does not offer a justifiable performance advantage.

Figure 2 shows a speedup graph for parallelization of simulations (measured using getCurrentTime).

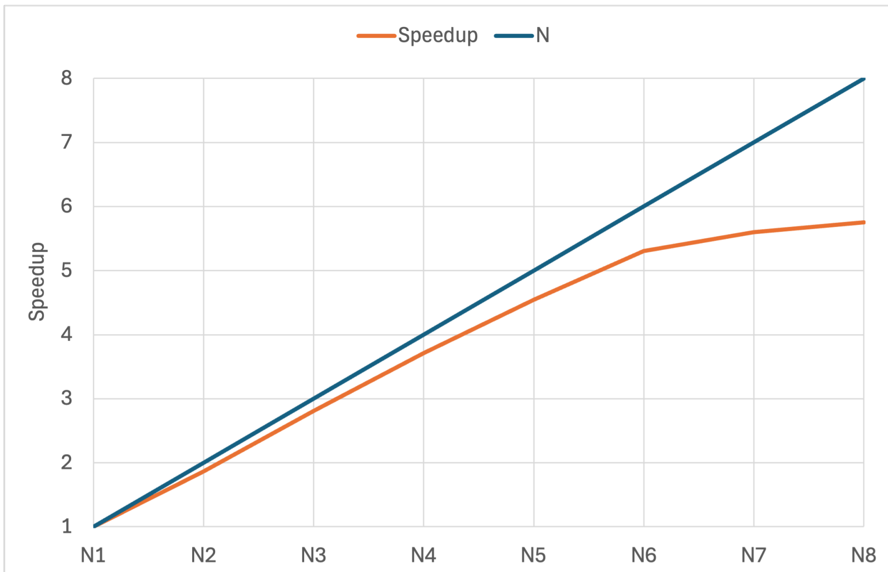

Figure 2: Basic Parallel Speedup

3.2 Simulation Chunking

In efforts to optimise our parallel evaluator, we created an implementation using manual chunking of game simulations. The simulations are split into a specified chunk size and parMap is called across chunks instead of all simulations. We performed testing for different chunk sizes as shown below:

Chunk Size	Average Runtime (s)
512	1.199
256	1.185
128	1.133
64	1.114
32	1.143
16	1.092
8	1.135
4	1.088
2	0.945

The apparent downward trend in runtime as chunk size decreases suggests that large chunk size is not an effective method of optimization. However, using a chunk size of 2 does appear to perform slightly better than the pure parMap implementation.

4 Thread Analysis

4.1 Simple parMap

```
Parallel GC work balance: 14.70% (serial 0%, perfect 100%)

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

SPARKS: 143360 (143360 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

INIT    time    0.005s  (  0.007s elapsed)
MUT     time    6.997s  (  0.936s elapsed)
GC      time    0.044s  (  0.033s elapsed)
EXIT    time    0.002s  (  0.010s elapsed)
Total   time    7.049s  (  0.987s elapsed)
```

As shown above, the basic parMap implementation creates a very large amount of sparks. However, all sparks are converted and speedup is roughly 7.14 (based on elapsed versus total CPU time), which is almost optimal for 8 cores. Additionally, the time spent on Garbage Collection appears to be insignificant, so we did not consider it to be a key point for optimization.

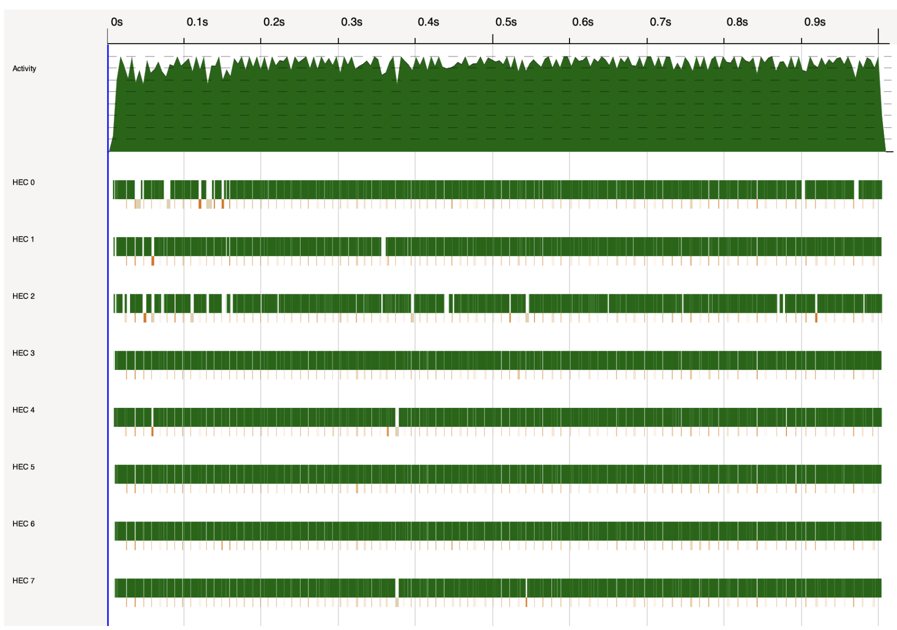

Figure 3: Pure parMap implementation

4.2 Chunk size 64

```
Parallel GC work balance: 38.61% (serial 0%, perfect 100%)

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

```

SPARKS: 2240 (2108 converted, 0 overflowed, 0 dud, 42 GC'd, 90 fizzled)

INIT    time    0.005s  (  0.004s elapsed)
MUT     time    6.800s  (  1.131s elapsed)
GC      time    0.013s  (  0.012s elapsed)
EXIT    time    0.001s  (  0.001s elapsed)
Total   time    6.819s  (  1.148s elapsed)
```

Our chunked parallel implementation creates far fewer sparks at a chunk size of 64 and spends less time on garbage collection. Though, its performance at this size is notably worse, which is made apparent by the poor load balancing shown in threadscope.

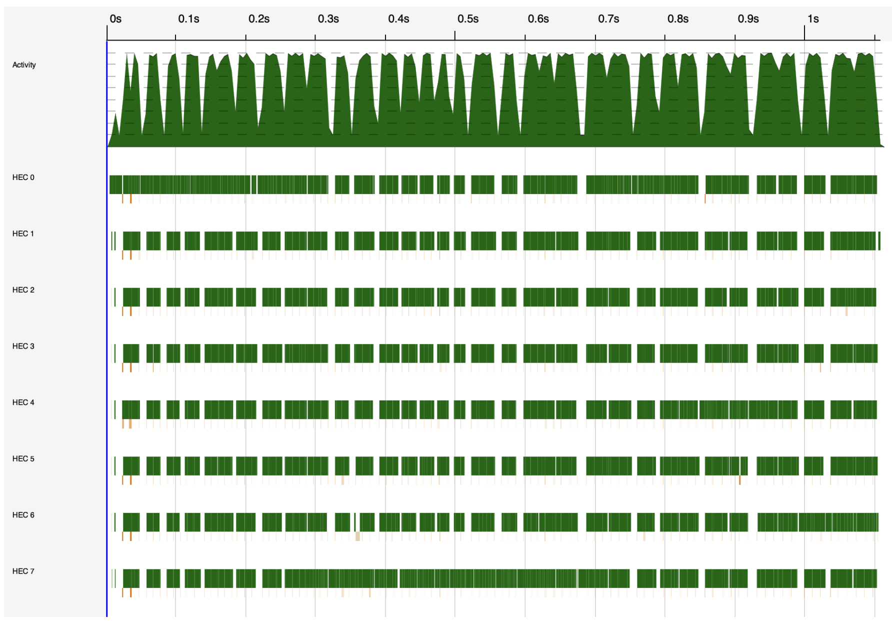

Figure 4: Chunk size 64

4.3 Chunk Size 2

```
Parallel GC work balance: 20.78% (serial 0%, perfect 100%)

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

SPARKS: 71680 (71667 converted, 0 overflowed, 0 dud, 0 GC'd, 13 fizzled)

INIT    time    0.006s  (  0.005s elapsed)
MUT     time    6.997s  (  0.925s elapsed)
GC      time    0.026s  (  0.018s elapsed)
EXIT    time    0.002s  (  0.010s elapsed)
Total   time    7.031s  (  0.958s elapsed)
```

Chunking with size 2 appears to provide the best load balancing, as shown in Figure 4. This implementation has roughly the same MUT time as the pure parMap one. However it creates half as many sparks and spends roughly half as much time on garbage collection, giving it a slight runtime advantage.

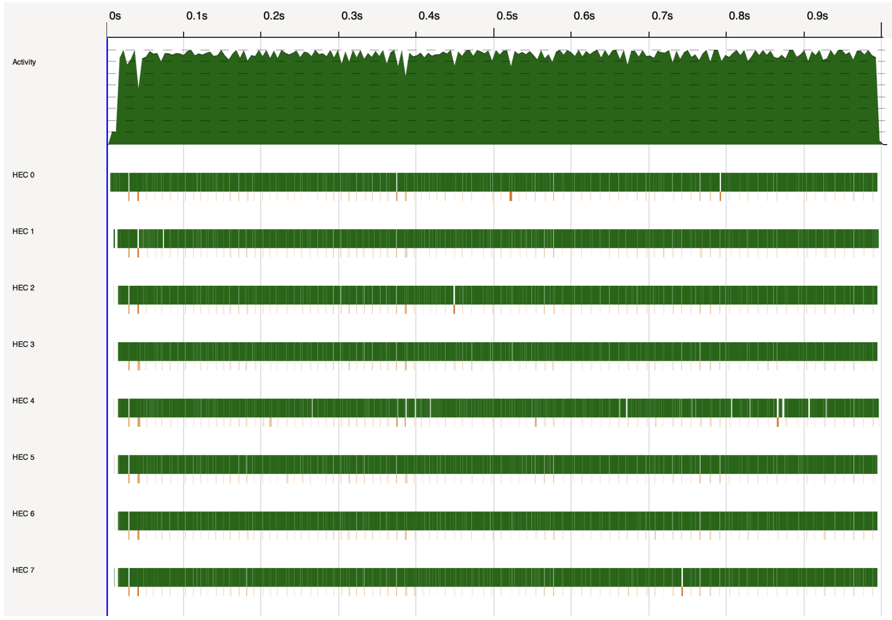

Figure 5: Chunk size 2

5 Final Implementation

5.1 Performance

Given that our chunked parallel implementation with size 2 demonstrated the best initial performance, we chose it as our final implementation.

Further optimization was offered by adding the -A32m flag, bringing the garbage collection time from 18ms to 5ms. This improvement was discovered in a project report from 2024: "A* Search For TSP."

Figure 6 shows a speedup graph for this implementation:

N	Runtime (s)	Speedup
1	5.71	1
2	3.04	1.88
3	2.01	2.84
4	1.51	3.77
5	1.22	4.66
6	1.03	5.56
7	0.98	5.83
8	0.92	6.17

5.2 Further Optimization

While this implementation did offer significant speedup at 8 cores, there are a few areas that could be further improved upon. For one, tests were performed on an M1 Pro chip, which has an asymetric core setup of 6 "Performance" cores and 2 "Efficiency" cores. This could explain the increased gap between speedup and N apparent at N7 and N8. Additionally, our implementation uses a simple array for the board and Int's for the column indices. Using more efficient data structures as well as better methods of applying moves and win checking

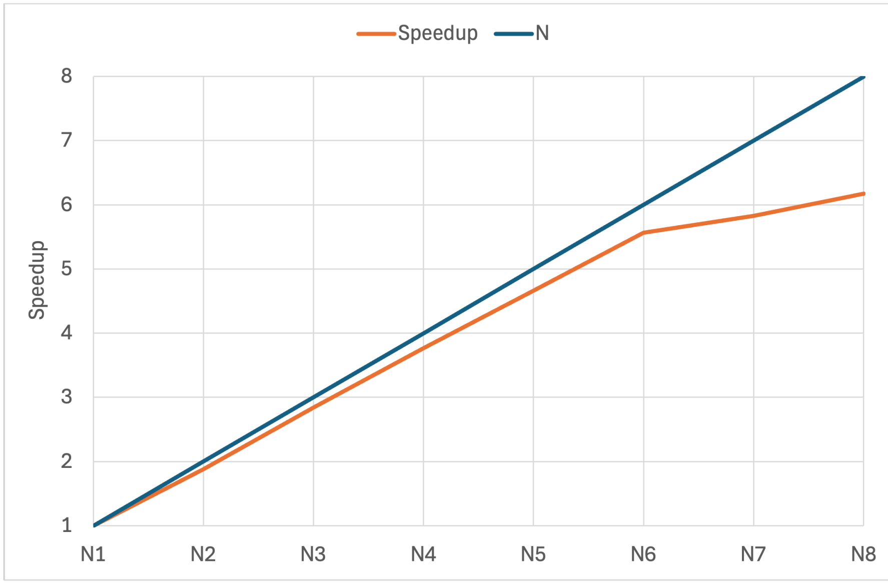

Figure 6: Chunk Size 2 Parallel Speedup

would likely make our implementation much faster.

5.3 Playing Against Evaluator

We created an interactive Connect-4 game in the command line to allow for play against the evaluator. It uses the size 2 chunked implementation with 2048 simulations. We found its moves to be very quick and difficult to beat in most circumstances. Figure 7 shows some screenshots (the evaluator is Yellow).

6 Code

https://github.com/gln2109/montecarlo-c4

6.1 Board.hs

```
{-# LANGUAGE DeriveGeneric #-}
module Board where

import Data.Maybe
import GHC.Generics (Generic)
import Control.DeepSeq
import System.Random

data Player = Red | Yellow deriving (Eq, Show, Generic)
instance NFData Player

type Board = [[Maybe Player]]

-- set board size
boardRows, boardCols :: Int
boardRows = 6
boardCols = 7

-- helper function for turn swap
```

Figure 7: Connect-4 Gameplay

```
otherPlayer :: Player -> Player
otherPlayer Red = Yellow
otherPlayer Yellow = Red

 -- generate empty board
emptyBoard :: Board
emptyBoard = replicate boardRows (replicate boardCols Nothing)

 -- return array of open cols
availableMoves :: Board -> [Int]
availableMoves board = [col | col <- [0..boardCols-1], isNothing (board !! 0 !! col)]

-- return new Board with move applied
applyMove :: Board -> Player -> Int -> Board
applyMove board player col =
    case [row | row <- [0..boardRows-1], isNothing (board !! row !! col)] of
        []        -> board
        emptyRows -> let row = maximum emptyRows
                         newRow = take col (board !! row) ++ [Just player] ++ drop (col+1) (board !! row)
                     in take row board ++ [newRow] ++ drop (row+1) board

-- recursively apply an array of moves, alternating players
applyMoveList :: Board -> Player -> [Int] -> Board
applyMoveList board _ [] = board
```

```haskell
applyMoveList board player (col:cols) = applyMoveList (applyMove board player col) (otherPlayer player) cols

-- return a string representation of the board
boardString :: Board -> String
boardString board =
    unlines [
        concat [playerChar (board !! row !! col) ++ " " | col <- [0..boardCols-1]]
        | row <- [0..boardRows-1]
    ]
    where
        playerChar Nothing = "."
        playerChar (Just Red) = "R"
        playerChar (Just Yellow) = "Y"

-- return a winning player if one exists
checkWin :: Board -> Maybe Player
checkWin board = checkCells [(row,col) | row <- [0..boardRows-1], col <- [0..boardCols-1]]
    where
        checkFour row col drow dcol =
            let lastRow = row + 3*drow
                lastCol = col + 3*dcol
            in if lastRow < 0 || lastRow > boardRows-1 || lastCol< 0 || lastCol > boardCols-1 then Nothing
            else case board !! row !! col of
                 Nothing     -> Nothing
                 Just player ->
                    let otherCells = [(board !! (row + d*drow) !! (col + d*dcol)) | d <-[1..3]]
                    in if all (== Just player) otherCells then Just player else Nothing

        checkDirs :: (Int,Int) -> [(Int,Int)] -> Maybe Player
        checkDirs _ [] = Nothing
        checkDirs cell (dir:dirs) =
            case checkFour (fst cell) (snd cell) (fst dir) (snd dir) of
                Nothing     -> checkDirs cell dirs
                Just player -> Just player

        checkCells :: [(Int, Int)] -> Maybe Player
        checkCells [] = Nothing
        checkCells (cell:cells) =
            case checkDirs cell [(1,0),(0,1),(1,1),(1,-1)] of
                Nothing     -> checkCells cells
                Just player -> Just player

-- run a simulation and return the winner
simulate :: Board -> Player -> StdGen -> Maybe Player
simulate board player gen =
    case checkWin board of
        Just winner -> Just winner
        Nothing ->
            let moves = availableMoves board
            in if moves == [] then Nothing
            else
                let (moveIndex, newGen) = randomR (0, length moves - 1) gen
                    newBoard = applyMove board player (moves !! moveIndex)
                in simulate newBoard (otherPlayer player) newGen
```

6.2 EvalSeq.hs

```haskell
module EvalSeq (bestMoveSeq) where
```

```

import Board
import System.Random
import Data.Function (on)
import Data.List

-- count simulation wins for each move
evalMove :: Board -> Player -> Int -> Int -> (Int, Int)
evalMove board player simulations move =
    let gens = [mkStdGen s | s <- [1..simulations]]
        newBoard = applyMove board player move
        results = map (simulate newBoard (otherPlayer player)) gens
    in (move, length (filter (== Just player) results))

-- return the move with the most wins
bestMoveSeq :: Board -> Player -> Int -> Int
bestMoveSeq board player simulations =
    let moves = availableMoves board
        winCounts = map (evalMove board player simulations) moves
    in (fst (maximumBy (on compare snd) winCounts))
```

6.3 EvalPar.hs

```
module EvalPar (bestMovePar) where

import Board
import System.Random
import Data.Function (on)
import Data.List
import Control.Parallel.Strategies

-- count simulation wins for each move
evalMove :: Board -> Player -> Int -> Int -> (Int, Int)
evalMove board player simulations move =
    let gens = [mkStdGen s | s <- [1..simulations]]
        newBoard = applyMove board player move
        results = parMap rdeepseq (simulate newBoard (otherPlayer player))
            gens
    in (move, length (filter (== Just player) results))

-- return the move with the most wins
bestMovePar :: Board -> Player -> Int -> Int
bestMovePar board player simulations =
    let moves = availableMoves board
        winCounts = map (evalMove board player simulations) moves
    in (fst (maximumBy (on compare snd) winCounts))
```

6.4 EvalChunk.hs

```
module EvalChunk (bestMoveChunk) where

import Board
import System.Random
import Data.Function (on)
import Data.List
import Control.Parallel.Strategies

-- run a chunk of simulations for a given move
simulateChunk :: Board -> Player -> Int -> [StdGen] -> Int
simulateChunk board player move gens =
    let newBoard = applyMove board player move
        results = map (simulate newBoard (otherPlayer player)) gens
```

```haskell
    in length (filter (== Just player) results)

-- split list into length n chunks
splitList :: [a] -> Int -> [[a]]
splitList [] _ = []
splitList li n = take n li : splitList (drop n li) n

-- count simulation wins for each move
evalMove :: Board -> Player -> Int -> Int -> Int -> (Int, Int)
evalMove board player simulations chunkSize move =
    let gens = [mkStdGen s | s <- [1..simulations]]
        genChunks = splitList gens chunkSize
        chunkWins = parMap rdeepseq (simulateChunk board player move) genChunks
        totalWins = sum chunkWins
    in (move, totalWins)

-- return the move with the most wins
bestMoveChunk :: Board -> Player -> Int -> Int -> Int
bestMoveChunk board player simulations chunkSize =
    let moves = availableMoves board
        winCounts = map (evalMove board player simulations chunkSize) moves
    in (fst (maximumBy (on compare snd) winCounts))
```

6.5 Bench.hs

```haskell
module Main where

import Board
import EvalSeq
import EvalPar
import EvalChunk
import Control.Monad (forM)
import Data.Time.Clock

testBoards :: [(Board, Player)]
testBoards = [(applyMoveList emptyBoard Red [3,3,2,4,1,4], Red),
              (applyMoveList emptyBoard Red [3,0,2,4,1,4,2], Yellow),
              (applyMoveList emptyBoard Red [3,0,2,4,1,4,3,4], Red),
              (applyMoveList emptyBoard Red [3,0,2,4,1,4,3,4,4,0,1], Yellow),
              (applyMoveList emptyBoard Red [3,0,2,4,1,4,3,4,4,0,1,2], Red)]

main :: IO ()
main = do
    putStrLn "Test Boards:\n"
    _ <- forM testBoards (\(board, player) -> do putStrLn (boardString board ++ "Player: " ++ (show player) ++ "\n"))
    let sims = 4096

    putStrLn "Sequential"
    start1 <- getCurrentTime
    moves1 <- forM testBoards (\(board, player) -> return (bestMoveSeq board player sims))
    putStrLn ("Moves: " ++ show moves1)
    end1 <- getCurrentTime
    putStrLn ("Time: " ++ show (diffUTCTime end1 start1))

    putStrLn "\nBasic Parallel"
    start2 <- getCurrentTime
    moves2 <- forM testBoards (\(board, player) -> return (bestMovePar board player sims))
    putStrLn ("Moves: " ++ show moves2)
    end2 <- getCurrentTime
    putStrLn ("Time: " ++ show (diffUTCTime end2 start2))
```

```

    putStrLn "\nChunked Parallel Size 256"
    start3 <- getCurrentTime
    moves3 <- forM testBoards (\(board, player) -> return (bestMoveChunk board player sims 256))
    putStrLn ("Moves: " ++ show moves3)
    end3 <- getCurrentTime
    putStrLn ("Time: " ++ show (diffUTCTime end3 start3))

    putStrLn "\nChunked Parallel Size 2"
    start4 <- getCurrentTime
    moves4 <- forM testBoards (\(board, player) -> return (bestMoveChunk board player sims 2))
    putStrLn ("Moves: " ++ show moves4)
    end4 <- getCurrentTime
    putStrLn ("Time: " ++ show (diffUTCTime end4 start4))
```

6.6 Play.hs

```
module Main where

import Board
import EvalChunk
import Text.Read (readMaybe)

getMove :: Board -> IO Int
getMove board = do
    putStrLn "Enter move (1-7):"
    move <- getLine
    case readMaybe move :: Maybe Int of
        Just num -> do
            if notElem (num-1) (availableMoves board) then do
                putStrLn "Invalid move."
                getMove board
            else return num
        Nothing -> do
            putStrLn "Invalid move."
            getMove board

playGame :: Board -> Player -> IO ()
playGame board player = do
    putStrLn ("\n" ++ show player ++ "'s turn:")
    putStrLn "1 2 3 4 5 6 7"
    putStrLn (boardString board)
    case checkWin board of
        Just winner -> putStrLn (show winner ++ " wins!")
        Nothing ->
            if (availableMoves board) == [] then putStrLn "Draw."
            else do
                if player == Red then do
                    playerMove <- getMove board
                    let playerBoard = applyMove board Red (playerMove-1)
                    playGame playerBoard Yellow
                else do
                    let evalMove = bestMoveChunk board Yellow 2048 2
                        evalBoard = applyMove board Yellow evalMove
                    putStrLn ("Yellow move: " ++ (show (evalMove+1)))
                    playGame evalBoard Red

main :: IO ()
main = do
    playGame emptyBoard Red
```