# Parallelizing Othello

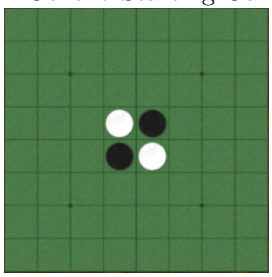
Carly Kiang (tk2990), Siying Ding (sd3609)

December 2025

# 1 Introduction

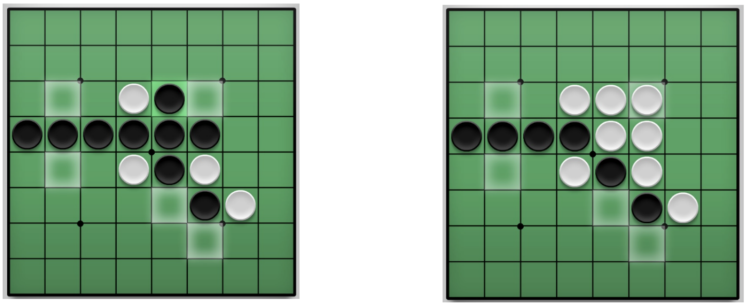
Othello, also known as Reversi, is a classic strategy board game played on an 8×8 grid between two players, traditionally represented by black and white discs. The game starts with four discs arranged in a square at the center of the board, two of each color. Players then take turns placing a disc of their color on the board with the goal of capturing their opponent's discs.

Figure 1: Othello Starting Game Board



The position a new disk is placed must abide by two rules; we will refer to them as the adjacency and outflanking rules. For the adjacency rule, a disc can only be placed on an empty square that is adjacent—horizontally, vertically, or diagonally—to at least one of the opponent's discs. In addition, to follow the outflanking rule, the placement of the new disk must create at least one straight line in which the newly placed disc flanks one or more of the opponent's discs between itself and another disc of the player's color. A disc flanks one or more of the opponent's discs if it is placed on the board so that there is a straight line (horizontal, vertical, or diagonal) of one or more opponent discs between the new disc and another disc of the player's color. All the opponent's discs in that line are considered flanked and are flipped to the player's color.

Figure 2: Game boards before (left) and after (right) player with white disc plays a move



# 2 Sequential Implementation

## 2.1 Game Setup

The Othello game board is an 8 by 8 grid where players will either place a white or black disk. To represent these components in Haskell, we decided to use a 2D Int array for the board, where a value of 0 would indicate an empty space, a value of 1 would represent a disc placed by player 1, and a value of 2 would represent a disc placed by player 2. An example of how the starting board would be represented is shown below.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 2 | 0 | 0 | 0 |
| 0 | 0 | 0 | 2 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

To improve code readability, we defined two type aliases that are used throughout method type signatures in our code. We also defined a data type BoardState to hold information about a current round in a game of Othello; the current game board and the current player.

```
-- BoardState
type Position = (Int, Int)

type Board = [[Int]]

data BoardState = BoardState {
    board      :: Board,
    curr_player :: Int
} deriving (Show)
```

To mimic the gameplay of a full Othello game, we wrote a `gameloop` method that would alternate between player 1 and player 2 turns until there was a winner or a tie.

## 2.2 Minimax

A common decision making strategy for optimal moves in two-player games like Othello is the minimax algorithm. Given an input of the current game board, the minimax algorithm constructs a decision tree of potential boards based on possible next moves. Each layer of the tree alternates between the maximizing player, which selects the highest possible board score, and the minimizing opponent, which selects the lowest possible board score. An optimal move for the current player is selected after exploring this game tree.

To quantify the score of a board within the minimax algorithm, we wrote a combination of three heuristic functions for evaluating a given board: one that counts the number of discs on a board belonging to the current player, another that counts the number of available next moves for the current player, and a third heuristic that counts the number of corner discs belonging to the current player. This logic is encapsulated in the `evaluateBoard` method.

```
-- Method for calculating overall heuristic score of a given board
evaluateBoard :: BoardState -> Int
evaluateBoard bs = 2*(cornerHeuristic bs) + (mobilityHeuristic bs) + (discCountHeuristic bs)
```

The sequential minimax algorithm was written recursively. To mimic each layer in the game decision tree, list comprehension was used to apply all possible next moves to the recursive minimax calls. Details of this implementation can be seen in the code snippet below.

```
-- Minimax algorithm
miniMax :: BoardState -> Int -> Bool -> Int
miniMax bs depth isMaximizingPlayer
    | depth == 0 || null moves = evaluateBoard bs
    | isMaximizingPlayer = maximum [miniMax (updateTurn move bs) (depth-1) False | move <- moves]
    | otherwise        = minimum [miniMax (updateTurn move bs) (depth-1) True | move <- moves]
    where
```

```
      moves = getPossibleMoves bs
```

## 2.3 Minimax with Alpha-Beta Pruning

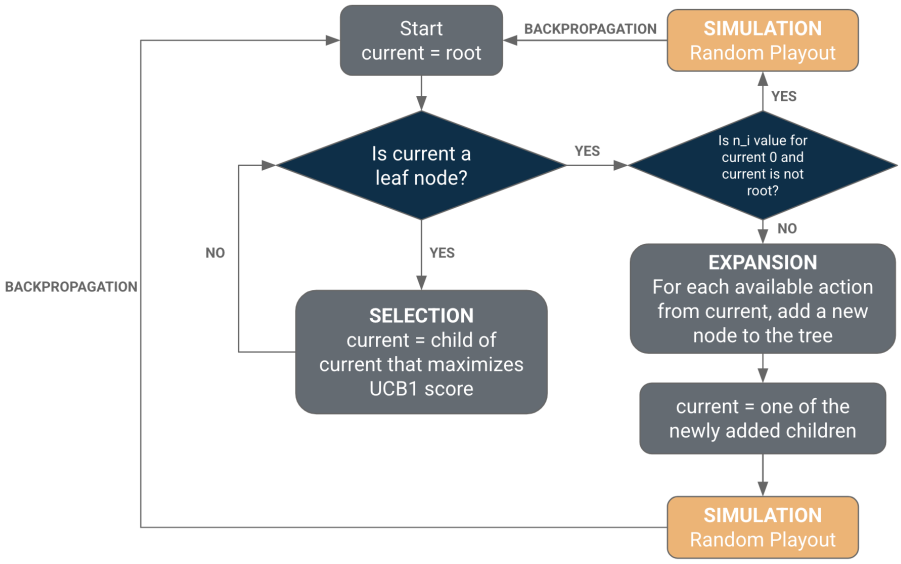
Alpha-Beta pruning is an optimization technique for short-circuiting the minimax algorithm. It prunes the decision tree by keeping track of a maximizing value $\alpha$ and a minimizing value $\beta$, and ignoring branches which are guaranteed to have worse scores that previously searched branches. In the Haskell code, this was achieved by writing local functions that either kept track of running $\alpha$ values (for maximizing players) or running $\beta$ values (for minimizing players). The code implementation references a Reddit thread [2].

```
-- Minimax algorithm with Alpha Beta Pruning
miniMaxAlphaBeta :: BoardState -> Int -> Bool -> Int -> Int -> Int
miniMaxAlphaBeta bs 0 _ _ _ = evaluateBoard bs

miniMaxAlphaBeta bs depth True alpha beta = go alpha 0 [updateTurn move bs | move <-
    getPossibleMoves bs]
  where
    go a v (s:ss) = let v' = max v (miniMaxAlphaBeta s (depth - 1) False a beta)
                    in  if v' >= beta then v'
                        else go (max a v') v' ss
    go _ v [] = v

miniMaxAlphaBeta bs depth False alpha beta = go beta 10000 [updateTurn move bs | move <-
    getPossibleMoves bs]
  where
    go b v (s:ss) = let v' = min v (miniMaxAlphaBeta s (depth - 1) True alpha b)
                    in  if v' <= alpha then v'
                        else go (min b v') v' ss
    go _ v [] = v
```

## 2.4 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a simulation-based algorithm for selecting optimal moves in sequential decision-making problems, particularly in game playing such as Othello. Instead of exhaustively searching the entire tree representing the game states, MCTS incrementally builds a partial decision tree through an iterative process consisting of four phases: selection, expansion, simulation, and backpropagation. The algorithm balances exploration (evaluating previously unvisited or infrequently visited moves) and exploitation (favoring moves with high estimated value) by performing randomized simulations (playouts), to approximate the expected reward of each action. Rather than enumerating the entire game tree, MCTS focuses computational effort on the most promising regions. Each node in the tree maintains statistical metadata, including the number of visits and the accumulated reward, which are continuously updated to guide future search iterations. Figure 3 and explanation below provide a high-level overview of how one iteration of MTCS works [4]:

Figure 3: Overview of One Iteration of MCTS




1. **Selection**: Staring from the decision tree root, MCTS chooses a node to explore further using a particular formula. In our implementation, we used the UCB1 (Upper Confidence Bound 1) / UCT (Upper Confidence bounds applied to Trees) formula. The formula calculates the reward score for a given move/node in the MCTS search tree using parameters such as the average value of the node $v_i$ (total reward score $t$ divided by number of visits $n_i$), the number of parent visits ($N$), and the number of visits to the given node ($n_i$).

$$UCT(s_i) = \frac{t}{n_i} + \sqrt{2}\sqrt{\frac{\ln N}{n_i}} = v_i + \sqrt{2}\sqrt{\frac{\ln N}{n_i}}$$

2. **Expansion**: After obtaining a selected node from the Selection phase, if the node has been visited before ($n_i \neq 0$), we add a child node to the selected node for each available move from the selected node. We select one of these newly added child nodes to perform random playout. Otherwise, we directly perform random playout on the selected node from the Selection phase.

3. **Simulation**: Random game moves are simulated on the selected leaf node from previous phase until the game of Othello terminates. We calculate a reward score for the terminal state.

4. **Backpropagation**: The reward score from the simulation stage is backpropagated up the tree, from the selected leaf node all the way back to the root. At each ancestor node—including the parent, grandparent, and so on—the node's metadata is updated as follows: the visit count $n_i$ is incremented by 1 to reflect the traversal through this state, and the total reward $t$ is increased by the reward obtained from the terminal state. This process ensures that information from the simulation informs the evaluation of all nodes along the path, guiding future selection and expansion decisions.

A new data type `MCTSNode` was created to use within the Monte Carlo Tree Search algorithm. `MCTSNode` stores information such as the current BoardState, a list of children (which are also `MCTSNode`, allowing `MCTSNode` to recursively represent the entire MCTS search tree), the number of times this particular `MCTSNode` was visited, and the current reward score $t$ of the `MCTSNode`, and the move $m$ that leads to this state/node.

```
data MCTSNode = MCTSNode {
    state      :: BoardState,
    children   :: [MCTSNode],
    n_i        :: Int, -- Number of times visited
```

```
    t          :: Double, -- Total reward score
    m          :: Maybe Position -- The move that led to this state
} deriving (Show)
```

Since the sequential implementation for MCTS is relatively long, we will omit pasting the full code within the body of this report. However, the relevant code can be referenced in the appendix. To provide a high-level overview of the code logic, the following helper methods were written to modularize the code. These helper methods correspond to one of the four stages of Monte Carlo Tree search.

- **selection**: Select the best child to perform simulation based on UCT values
- **uctValue**: Calculate UCT value for a given MCTSNode
- **expansion**: Given an MCTSNode, for each available move, add a child node representing the move to the given MCTSNode
- **simulation**: Given an MCTSNode and a `StdGen` random generator, simulate a random playout until game termination
- **backpropagation**: Given a child MCTSNode and a reward value from the simulation stage, back-propagate the reward of terminal state up to the root and update ancestor nodes' metadata values along the way
- **runMCTSIteration**: Run a single iteration of MCTS, seeded by a `StdGen` random generator
- **runMCTS**: Run an arbitrary number of iterations of MCTS, as specified by the argument passed to `runMCTS`, with each iteration seeded by a different `StdGen` random generator to ensure true randomization and exploration

# 3 Parallel Implementation

## 3.1 Minimax: Top Level Parallelism

One method of parallelism for the minimax algorithm is top level parallelism, where the first layer of children board states after the root are evaluated in parallel. This was achieved using `parList` and `rdeepseq` on the first recursive call to the `miniMax` method. `parList` evaluates each element in a list according to the given strategy, `rdeepseq`. `rdeepseq` is a strategy where the argument is fully evaluated. In other words, the each parallel child will fully evaluate all of it's sub-branches in the game decision tree.

Our code implementation performs the first call to `miniMax` within the `gameLoop` method. This means that `parList` can be directly used in the initial call, and not within the `miniMax` implementation itself.

```
let possibleMovesWithScores = [ (m, miniMax (updateTurn m bs) 3 True) | m <- possibleMoves ]
    `using` parList rdeepseq
```

## 3.2 Minimax: Top Level Chunked Parallelism

Another similar method of parallelism is top level chunked parallelism. The first layer of children board states after the root are evaluated in parallel chunks. This was achieved using `parBuffer` and `rdeepseq` on the first recursive call to the `miniMax` method. `parBuffer` takes in two arguments, the size of a chunk $n$ and the strategy that each chunk uses. For similar reasons as the top level parallel implementation, `rdeepseq` was the strategy we used. After some trial and error with different chunk sizes, the chunk size $n$ we passed into `parBuffer` was 150.

```
let possibleMovesWithScores = [ (m, miniMax (updateTurn m bs) 3 True) | m <- possibleMoves ]
    `using` parBuffer 8 rdeepseq
```

### 3.3 Minimax with Alpha-Beta Pruning

Similar parallelization methods from `miniMax` were applied to the minimax algorithm with Alpha-Beta pruning: `parList` and `parBuffer`. Since Alpha-Beta pruning is a sequential algorithm, the strategy attached to both `parList` and `parBuffer` was still `rdeepseq`.

As with `miniMax`, the top level recursive call in `gameLoop` was parallelized. Below is the code snippet corresponding to the top level parallelized `miniMaxAlphaBeta`:

```
let possibleMovesWithScores = [ (m, miniMaxAlphaBeta (updateTurn m bs) 3 True 0 100000) | m <-
    possibleMoves ] `using` parList rdeepseq
```

Below is the code snippet corresponding to the chunked top level parallelized `miniMaxAlphaBeta`:

```
let possibleMovesWithScores = [ (m, miniMaxAlphaBeta (updateTurn m bs) 3 True 0 100000) | m <-
    possibleMoves ] `using` parBuffer150 rdeepseq
```

### 3.4 MCTS: Top Level Parallel MCTS

To parallelize Monte Carlo Tree Search, we choose to use root parallelization. The idea behind root parallelization is that multiple independent MCTS are launched from the given root node in parallel, each performing iterations of MCTS and building the search tree in parallel [3]. Since the trees are built in parallel, the threads do not share information with each other, and we do not need to worry about potential race conditions that would have arisen if multiple threads are modifying the same MTCS tree instance. After running a fixed number of MCTS iterations, the root's children from separated MCTS trees are merged with their corresponding clones (e.g. adding $n_i$ and $t$ of the child node with move $m_i$ together across all MCTS trees that are built in parallel), and the best move from the root is selected based on this merged tree. We choose root parallelization as a parallelization technique because it can be incorporated smoothly into our original sequential implementation of MCTS since we already have helper functions like `runMCTS` that can run MCTS for an arbitrary number of iterations. Each thread can independently execute these functions, requiring minimal changes to the existing code.

A simple way to incorporate root parallelization for MCTS is splitting the number of iterations evenly across the number of threads. To do so, we write a helper function called `runMCTSParallel` to distribute 1000 MCTS iterations evenly across n available threads, as shown below.

```
runMCTSParallel :: MCTSNode -> Int -> StdGen -> Int -> MCTSNode
runMCTSParallel root iterations gen threadNum = mergedRoot
    where
        (q, r) = iterations `divMod` threadNum
        -- For simplicity, if iterations is not divisible by threadNum, we just give every thread
            an extra iteration instead of figuring out how to distribute the remainder r (which
            may incur more overhead for distribution)
        -- This also simplifies the code for splitting the random generators and for calling parMap
            rdeepseq
        iterationsPerThread = if r == 0 then q else q + 1
        gens = take (iterationsPerThread * threadNum) $ iterate (snd . split) gen
        gensGrouped = chunksOf iterationsPerThread gens
        updatedRoots = parMap rdeepseq (\g -> runMCTS root iterationsPerThread g) gensGrouped
        mergedRoot = mergeMCTSRuns (tail updatedRoots) (head updatedRoots)

mergeMCTSRuns :: [MCTSNode] -> MCTSNode -> MCTSNode
mergeMCTSRuns [] root = root
mergeMCTSRuns (x:xs) root = mergeMCTSRuns xs mergedRoot
    where
        mergedRoot = root {
```

```
            n_i = n_i root + n_i x,
            t = t root + t x,
            children = mergeChildren (children root) (children x)
        }

mergeChildren :: [MCTSNode] -> [MCTSNode] -> [MCTSNode]
mergeChildren (x:xs) ys = mergedChild : mergeChildren xs ys
    where
        move = m x
        -- find corresponding child in ys based on move
        y = head $ filter (\child -> m child == move) ys
        mergedChild = x {
            n_i = n_i x + n_i y,
            t = t x + t y,
            m = move, -- both children should have same move
            children = [] -- We don't need to go deeper since we only care about root's children
                for MCTS decision making
        }

let mctsRoot = runMCTSParallel rootNode 1000 gen n -- n is the number of threads
```

`mctsRoot` will be the merged tree containing the aggregated metadata of each of the root's child nodes across trees that are built in parallel.

### 3.5 MCTS: Chunked Top Level Parallel MCTS

Another way to incorporate root parallelization, at a more granular level, is to split the total number of MCTS iterations into chunks of 10 iterations each. Then we use `parMap` to evaluate each of these chunks in parallel and use `rdeepseq` force these chunks to be fully evaluated instead of being left as thunks due to lazy evaluation. By splitting the task into smaller chunks of 10 iterations each, it allows idle threads that finish early to pick up additional tasks, improving load balancing and parallelization efficiency. This contrasts with the previous approach of using larger chunks (chunk size = `iterationsPerThread` in code snippet above), where a thread assigned a particularly time-consuming chunk could delay overall execution while other threads remain idle, waiting for it to finish.

```
runMCTSParallel :: MCTSNode -> Int -> StdGen -> MCTSNode
runMCTSParallel root iterations gen = mergedRoot
    where
        gens = take iterations $ iterate (snd . split) gen
        gensGrouped = chunksOf 10 gens
        updatedRoots = parMap rdeepseq (\g -> runMCTS root 10 g) gensGrouped
        mergedRoot = mergeMCTSRuns (tail updatedRoots) (head updatedRoots)

let mctsRoot = runMCTSParallel rootNode 1000 gen
```

### 3.6 MCTS: Parallelize Tree Merging Process

Aside from root parallelization, we notice that the process of merging different trees, as shown in `mergeMCTSRuns`, is performed sequentially. So we also parallelize `mergeMCTSRuns` in hope of gaining more speedup. To parallelize `mergeMCTSRuns`, we use a similar chunking technique (as shown above), where we split the parallely-built trees into chunks of 10 trees each and use `parMap rdeepseq` to distribute the work of merging trees across available threads.

```
mergeMCTSRunsParallel :: [MCTSNode] -> MCTSNode
mergeMCTSRunsParallel [] = error "Something is wrong with mergeMCTSRunsParallel" -- Should not
    really happen
```

```
mergeMCTSRunsParallel [x] = x
mergeMCTSRunsParallel xs = mergeMCTSRunsParallel mergedChunks
    where
        chunks = chunksOf 10 xs
        mergedChunks = parMap rdeepseq (\c -> mergeMCTSRuns (tail c) (head c)) chunks

-- Same runMCTSParallel implementation from previous code snippet, but replace
-- mergedRoot = mergeMCTSRuns (tail updatedRoots) (head updatedRoots)
-- with
mergedRoot = mergeMCTSRunsParallel updatedRoots
```

# 4 Performance Comparisons

## 4.1 Benchmarking

To benchmark the runtime of our various parallelized implementations of Othello, we recorded the time elapsed for a given thread configuration, averaged over 100 trials. minimax algorithms were run with a tree depth of 3 and MCTS algorithms were run with 1000 iterations. To ensure confidence in algorithm correctness, we also tracked the win-rate of the algorithms, which averaged to be around $85\% - 87\%$ for minimax algorithms and $93\% - 99\%$ for MCTS algorithms.

Each game of Othello starts with the same configuration, as shown in section 1. One trial consisted of a full `gameLoop`, where player 1 (the opponent) would select a random move from a list of available moves, and player 2 would run the Minimax of MCTS algorithms to find and play an optimal move. A trial concluded when the game of Othello finished.

All benchmarking experiments were ran on a laptop with the following specifications:

- **Brand**: Apple
- **Model**: M2
- **Cores**: 8
- **Hardware Threads**: 8

## 4.2 Minimax

Figure 4: Actual Speedup vs. Ideal Speedup for Top Level Parallelism (Left) and Chunked Top Level Parallelism (Right) Applied to Minimax

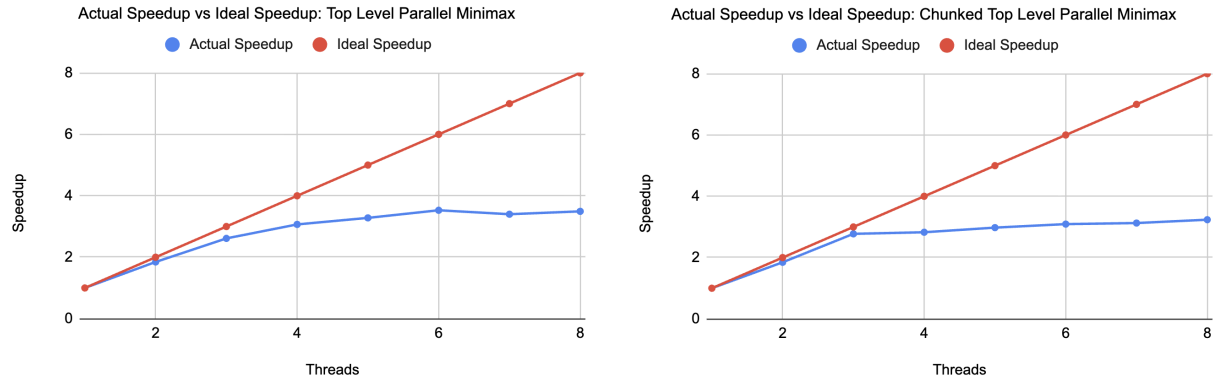


Table 1: Time Elapsed for Minimax Without Pruning: Top Level Parallelism

| Threads | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| Time Elapsed (seconds) | 3.455 | 1.872 | 1.322 | 1.126 | 1.053 | 0.979 | 1.016 | 0.989 |

Table 2: Time Elapsed for Minimax Without Pruning: Chunked Top Level Parallelism

| Threads | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| Time Elapsed (seconds) | 3.621 | 1.961 | 1.305 | 1.281 | 1.216 | 1.171 | 1.158 | 1.119 |

The performance of top level parallelism compared to chunked top level parallelism was relatively similar for the minimax algorithm. Actual speedup for both parallel implementations performs close to ideal speedup for two threads, but achieves less speedup for an increased number of threads. The peak actual speedup achieved for top level parallel minimax was 3.529 times at 6 threads, and the peak actual speedup achieved for chunked top level parallel minimax was 3.23 times at 8 threads. Top level parallelism had slightly better performance than chunked parallelism.

The Threadscope event logs for the top level parallel implementation 5 and the chunked top level parallel implementation 6 show similar performances, where all 8 threads spend a good portion of time running.

Figure 5: Threadscope Event Log of Top Level Parallel Minimax on 8 Cores

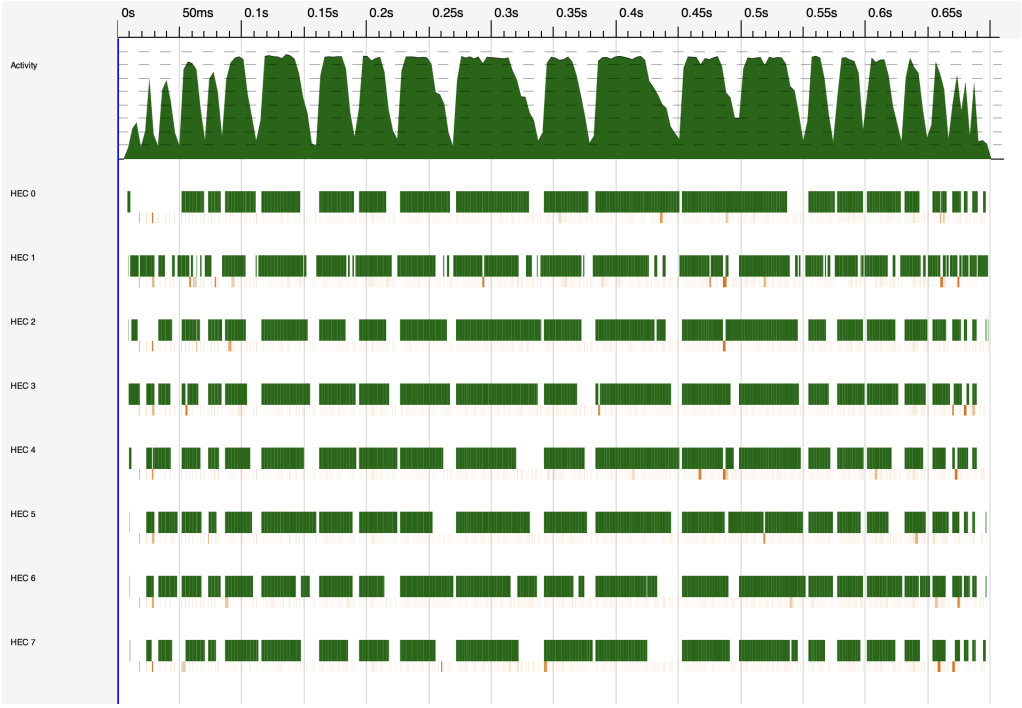
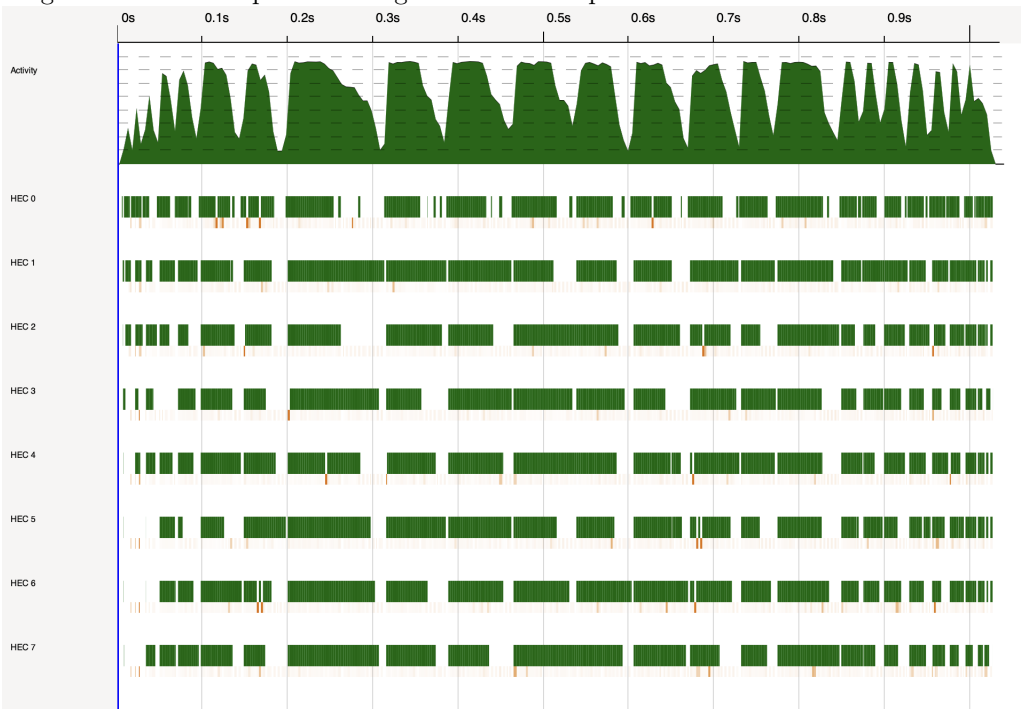


Figure 6: Threadscope Event Log of Chunked Top Level Parallel Minimax on 8 Cores




## 4.3 Minimax with Alpha-Beta Pruning

Figure 7: Actual Speedup vs. Ideal Speedup for Top Level Parallelism (Left) and Chunked Top Level Parallelism (Right) Applied to Minimax with Alpha-Beta Pruning


The chunked top level parallelism implementation performed slightly better than the top level parallelism implementation for minimax with alpha beta pruning. The peak actual speedup achieved for top level parallel implementation was 3.199 times at 7 threads, and the peak actual speedup achieved for chunked top level parallel implementation was 3.39 times at 7 threads.

For both parallel implementations, actual speedup continued to improve up until a thread count of 7, with the average speedup achieved by 8 threads being lower than the average speedup achieved by 7 threads. This suggests that a different parallelization strategy may be required to achieve better performance gains

for higher thread counts.

Examining the Threadscope event log for the top level parallel implementation 8 shows that when using 8 threads, there is a lot of idle time for each thread, and not all threads are being utilized equivalently. An improvement can be seen in the Threadscope event log for the chunked top level parallel implementation 9, where all 8 threads have increased time spent running. However, when comparing both threadscope event logs produced by minimax with the event logs produced minimax with alpha-beta pruning, we see that the performance of parllelized minimax is better; each thread has more runtime. A possible explanation for this limitation is that less work is needed per top-level child due to the optimization from pruning.

The actual speedup compared to ideal speedup for parallelized minimax with alpha beta pruning performs worse than the the actual speedup for parallelized minimax without pruning. However, there are improvements to the runtime itself (time elapsed in seconds) when using minimax with alpha beta pruning. Notably, the average time elapsed for minimax with alpha beta pruning running on 1 thread was around 0.8 seconds (as seen below in Tables 3 and 4). This was faster than the average time elapsed for parallelized minimax without pruning running on 8 threads (around 3.5 seconds, as seen in Tables 1 and 2).

Figure 8: Threadscope Event Log of Top Level Parallel Minimax with Alpha-Beta Pruning on 8 Cores

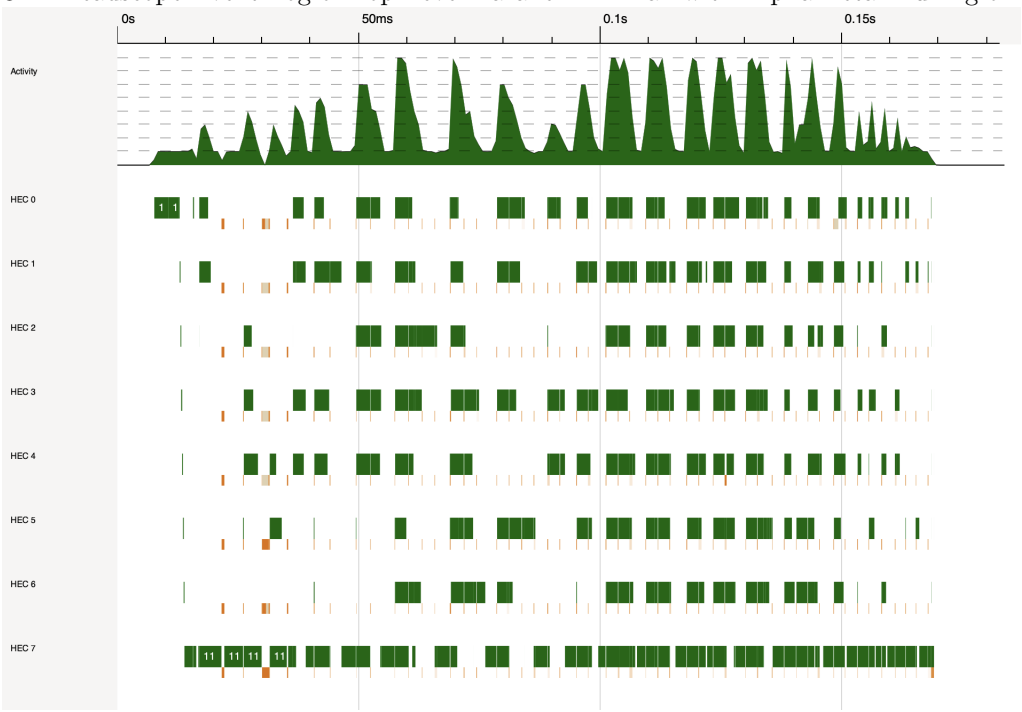


Figure 9: Threadscope Event Log of Chunked Top Level Parallel Minimax with Alpha-Beta Pruning on 8 Cores

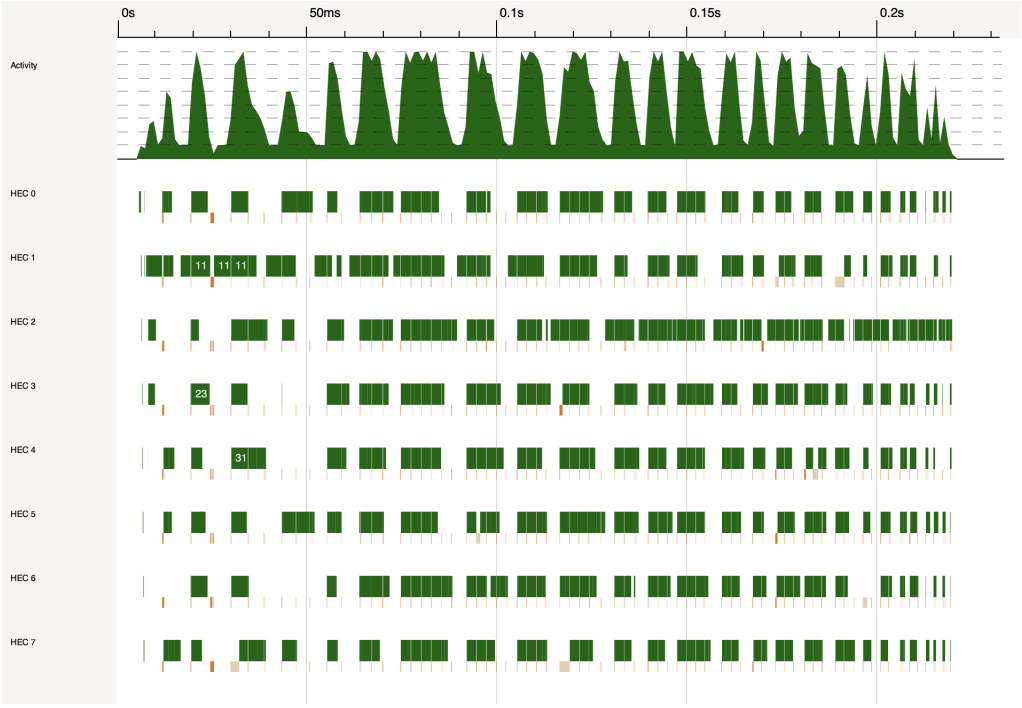



Table 3: Time Elapsed for Minimax with Pruning: Top Level Parallelism

| Threads | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| Time Elapsed (seconds) | 0.854 | 0.530 | 0.366 | 0.326 | 0.283 | 0.278 | 0.267 | 0.300 |

Table 4: Time Elapsed for Minimax with Pruning: Chunked Top Level Parallelism

| Threads | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| Time Elapsed (seconds) | 0.889 | 0.530 | 0.383 | 0.307 | 0.289 | 0.277 | 0.262 | 0.271 |

## 4.4 MCTS: Top Level Parallel MCTS

Figure 10 shows that the actual speedup of Top Level Parallel MCTS gradually plateaus at around 3 as the number of threads increases from 7 to 8. The Threadscope event log 11 shows sustained high CPU usage for the majority of the execution, and work is roughly evenly distributed across all available threads, which indicate effective parallelization of MCTS. However, it is worth noting that HEC0 seems to be more busy working than other threads, which are indicated by denser green bars. It may be that HEC0 is assigned a chunk of work that requires more computations than others (e.g. due to the `StdGen` random generators we seed the chunk of MCTS iterations with). This is the exact motivation behind Chunked Top Level Parallel MCTS in the next section, where we aim to improve load balancing and gain more speedup.

Figure 10: MCTS: Top Level Parallel MCTS

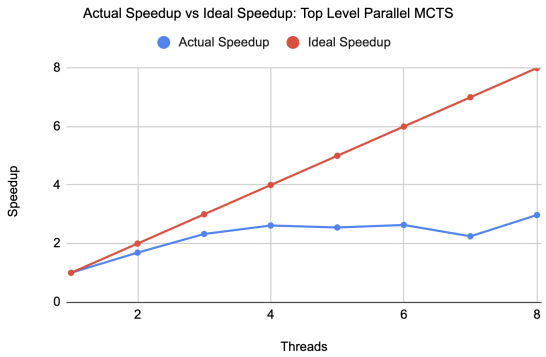



Table 5: Time Elapsed for MCTS: Top Level Parallel MCTS

| Threads | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| Time Elapsed (seconds) | 5.218 | 3.087 | 2.242 | 1.994 | 2.047 | 1.980 | 2.321 | 1.754 |

Figure 11: Threadscope Event Log of Top Level Parallel MCTS on 8 Cores


## 4.5 MCTS: Chunked Top Level Parallel MCTS

Figure 12 shows that by using chunked top level parallelism in MCTS, we are able to get better speedup, where the actual speedup curve aligns closely with the ideal speedup curve up until 4-thread configuration and it gradually plateaus at around 3.6 as the number of threads increases to 7 and 8. As we can see in the Threadscope figure 13, the works are evenly distributed across available cores with each HEC's green bars being of roughly similar density, and there are less and shorter idle times in each HEC. This validates the effectiveness of Chunked Top Level parallelization technique as well as the improved load balancing it

provides.
However, it is worth noting that even though we gained more speedup, the actual time elapsed of Chunked Top Level Parallel MCTS is actually slightly longer than Top Level Parallel MCTS. The potential reason is that smaller chunk size means more tasks, which incur overhead of scheduling. Since we are running the MCTS simulation for 1000 iterations, the overhead may actually outweigh the benefits we gain from improved load balancing. Nonetheless, the better speed up we see from figure 12 implies that Chunked Top Level Parallel MCTS will be a good parallelization of MCTS as the number of iterations increases (as discussion in Future Directions in Section 5), where the benefits of better load balancing may eventually outweigh scheduling overhead and we will be able to see Chunked Top Level Parallel MCT having shorter elapsed time than Top Level Parallel MCTS.

Figure 12: MCTS: Chunked Top Level Parallel MCTS

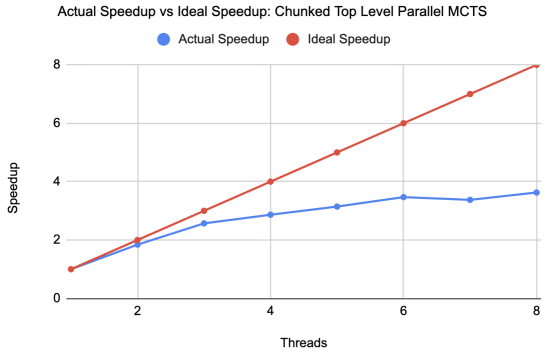
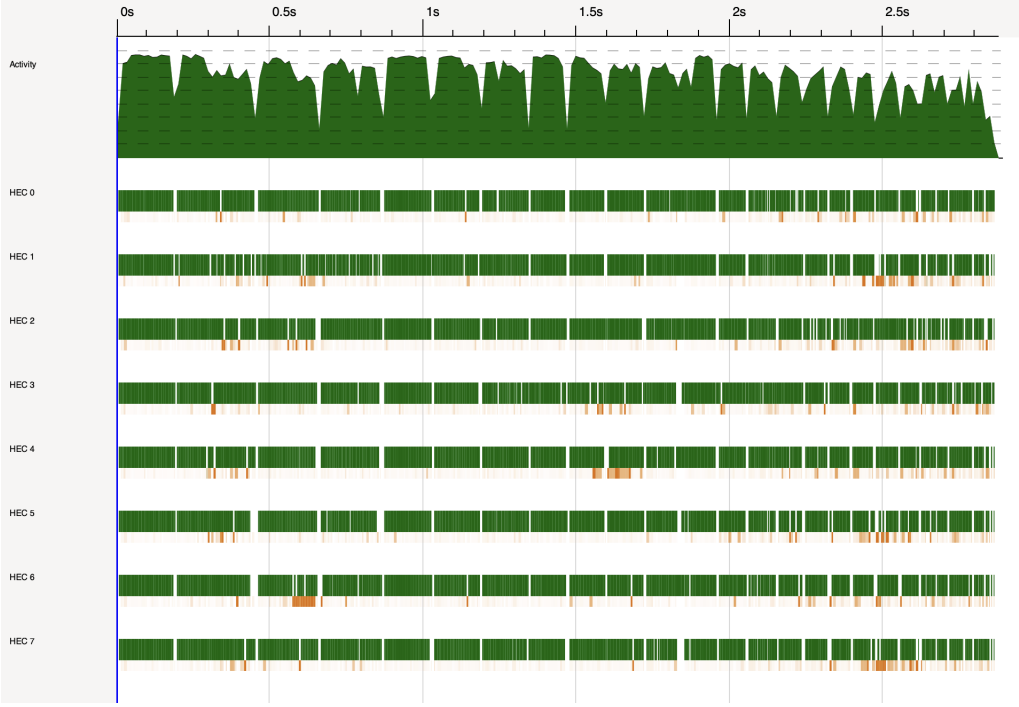



Table 6: Time Elapsed for MCTS: Chunked Top Level Parallel MCTS

| Threads | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| Time Elapsed (seconds) | 6.845 | 3.717 | 2.664 | 2.388 | 2.176 | 1.973 | 2.028 | 1.887 |

Figure 13: Threadscope of Chunked Top Level Parallel MCTS on 8 Cores




## 4.7 MCTS: Parallelize Tree Merging Process

The final attempt to improve speedup in the root parallelization of MCTS is to parallelize the sequential tree merging process on top of Chunked Top Level Parallel MCTS. As shown in figure 14, the actual speedup curve aligns closely with the ideal speedup curve up until 3-thread configuration. The speedup continues to increase as the number of threads increases, with final speedup of 3.7 at 8-thread configuration. The Threadscope figure 15 shows that works are evenly distributed across available cores.

Similar to the discussion in the previous section, even though MCTS with parallelized tree merging process has better speedup, the time elapsed is actually longer than sequential implementation because we are running MCTS for 1000 iterations at chunks of 10 iterations each, which means there are only 100 trees to merge at the end. The overhead of scheduling parallel merging of these trees actually actually outweighs the gains from parallelization. The better speed up we see from figure 14 (and the continuous increase in speedup as the number of threads increases) does imply that parallelized tree merging process can be helpful as the number of iterations increases (as discussion in Future Directions in Section 5), where the benefits of parallelization may eventually outweigh scheduling overhead and we will be able to see it having shorter elapsed time than sequential implementation.

Figure 14: MCTS: Parallelize Tree Merging Process

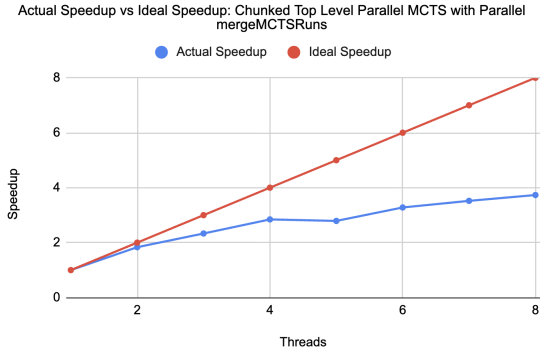
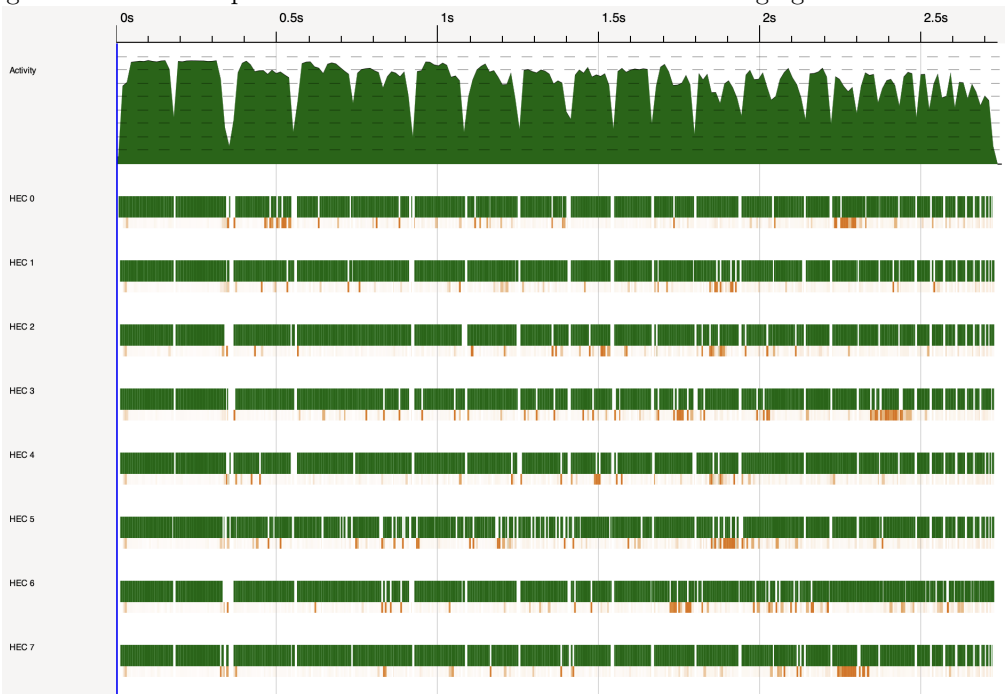



Table 7: Time Elapsed for MCTS: Parallelize Tree Merging Process

| Threads | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| Time Elapsed (seconds) | 6.953 | 3.791 | 2.981 | 2.444 | 2.492 | 2.12 | 1.975 | 1.863 |

Figure 15: Threadscope of MCTS with Parallelization of Tree Merging Process on 8 Cores




## 4.8 Comparing Minimax with MCTS

Overall, the actual speedup of two MCTS implementations was the better than all of the minimax implementations: the implementation with chunked root parallelization, and the implementation with parallelized tree merging. This suggests that in the context of Othello, MCTS is a more "parallelizable" algorithm compared to minimax. A potential explanation for this could be the fact that the top-level parallelization of minimax is constrained by the fact that there are only four choices of moves a player can make as their initial move, which bounds the number of independent subtrees that can be explored in parallel.

# 5 Future Directions

## 5.1 Othello Representation in Haskell

Our current representation of the board is a 2D array of `Int` objects. This leads to costly computations when trying to calculate possible moves for a player given a `BoardState`. An improvement could be to flatten the board representation into a 1D array of `Int` objects, and also keep auxillary data structures, such as a list of `Position` objects that correspond to the discs that belong to a player. Future work can focus on optimizing sequential implementation of Othello before moving on to incorporating parallelization techniques outlined in this paper.

## 5.2 Minimax

As a future extension, the Minimax with alpha-beta implementation can be improved further by incorporating more advanced alpha–beta pruning techniques, such as principal variation search [1]. By leveraging principal variation search, the search can establish tighter alpha-beta bounds earlier, leading to more effective pruning of suboptimal branches and reducing search space. This enhancement has the potential to significantly reduce the number of nodes evaluated, thereby improving search efficiency and reducing the runtime of Minimax algorithm.

## 5.3 Monte Carlo Tree Search

As future work, additional parallelization strategies beyond root parallelization for Monte Carlo Tree Search could be explored. In particular, the approaches described by Chaslot et al. [3], such as leaf parallelization (where multiple simulations are executed in parallel from a selected node) and tree parallelization (where multiple workers operate on a shared MTCS tree instance) are promising directions. It is worth noting that the later requires careful synchronization to maintain consistency of shared metadata, typically through mutex. Additionally, future experiments could vary the total number of MCTS iterations to identify the threshold at which chunked parallelization with parallelized tree merging process outperforms top level parallelization, providing further insight into the tradeoff between scheduling overhead and parallelization gain.

# 6 Conclusion

We explored Minimax (with and without alpha-beta pruning) and Monte Carlo Tree Search as two popular game search strategies for Othello. We parallelized both search strategies using top level parallelism and chunked top level parallelism. Parallelization of Minimax effectively reduces the runtime of the algorithm. MCTS, though more parallelizable than Minimax as indicated by higher speedup factor under root parallelization, incurs scheduling overhead that may outweigh the benefits of parallelization when the number of iterations is too small to amortize these overhead. As future works, it is worth exploring principal variation search for better pruning in Minimax with alpha-beta pruning and exploring other parallelization techniques of MCTS such as leaf parallelization to reduce the runtime further and leverage more parallelism.

# References


[1] `https://www.chessprogramming.org/Principal_Variation_Search`, 2024. ChessProgramming.org, accessed on December 14, 2025.

[2] `https://www.reddit.com/r/haskell/comments/1nigy1n/stumped_on_alpha_beta_pruning_in_haskell/`, 2025. Reddit thread, accessed on December 14, 2025.

[3] Mark H.M. Winands Guillaume M.J-B. Chaslot and H. Jaap van den Herik. Parallel monte–carlo tree search. In *Computers and Games*, volume 5131 of *Lecture Notes in Computer Science*, pages 60–71. Springer, 2008. PDF available at `https://dke.maastrichtuniversity.nl/m.winands/documents/multithreadedMCTS2.pdf`.

[4] John Levine. Monte carlo tree search. https://www.youtube.com/watch?v=UXW2yZndl7U, 2021. YouTube video, accessed on December 14, 2025.

# A Minimax Code Implementation

## A.1 othello-minimax-seq.hs

```
import System.IO()
import Data.Int()
import GHC.Base()

import System.Posix.Internals()
import Data.List (maximumBy)
import Data.Ord()
import Debug.Trace()
import System.Random(newStdGen, randomR)

{-
This file contains logic for sequential minimax without alpha beta pruning
The main function runs a playthrough of an Othello game, and prints
information for each turn. (Board state, available moves, selected move)

Commands to compile and run this program:
stack install random
stack ghc --package random -- -Wall -O2 -o othello othello-minimax-seq.hs
./othello
-}


-- Game types
type Position = (Int, Int)

type Board = [[Int]]

data BoardState = BoardState {
    board      :: Board,
    curr_player :: Int
} deriving (Show)

rows :: Int
rows = 8

cols :: Int
cols = 8

gameBoard :: Board
gameBoard = [[0 | _ <- [1..cols]] | _ <- [1..rows]] :: [[Int]]


updateBoardIndex :: Board -> Position -> Int -> Board
updateBoardIndex board (i,j) val =
    case splitAt i board of
        (prevRows, currRow : afterRows) ->
            case splitAt j currRow of
                (prevElems, _ : afterElems) ->
                    prevRows ++ [prevElems ++ [val] ++ afterElems] ++ afterRows
                (_, _) -> board
        (_, _) -> board


updateBoardIndexes :: Board -> [Position] -> [Int] -> Board
```

```
updateBoardIndexes board (x:xs) (y:ys) = updateBoardIndexes newBoard xs ys where
    newBoard = updateBoardIndex board x y
updateBoardIndexes board _ _ = board

{-
Initialize the gameboard to start configuration and set current player to 1
-}
initializeBoard :: BoardState
initializeBoard = BoardState {
    board = updateBoardIndexes gameBoard [(3,3),(4,4),(4,3),(3,4)] [1,1,2,2],
    curr_player = 1
}

{-
Printing for Board
-}
printBoard :: Board -> IO ()
printBoard board = mapM_ (putStrLn . unwords . map show) board


{-
Return the number at index (i,j) of the board
-}
getBoardVal :: Board -> Position -> Int
getBoardVal board (i,j) = (board !! i) !! j

{-
Defined 8-way directions for checking adjacency and flanking rules
-}
directions :: [Position]
directions = [(-1,-1), (-1,0), (-1,1), (0,-1), (0,1), (1,-1), (1,0), (1,1)]

{-
Check if a given position is within bounds of the board
-}
inBounds :: Position -> Bool
inBounds (x,y) = x >= 0 && x < rows && y >= 0 && y < cols

{-
Given a board state, return the possible next moves for the player
-}
getPossibleMoves :: BoardState -> [Position]
getPossibleMoves bs = [ (i,j) | i <- [0..rows-1],
                            j <- [0..cols-1],
                            getBoardVal (board bs) (i,j) == 0, -- spot is not taken yet
                            not (null (adjacencyAndFlankingCheck bs (i,j))) -- adjacency and
                                flanking rules are both met
                            ]

{-
Check if the adjacency and flanking conditions are both met for a given position
Return a list of directions where both conditions are met
If the list is non-empty, then the position is a valid move that satisfy both adjancy and
    flankinng rules
-}
adjacencyAndFlankingCheck :: BoardState -> Position -> [Position]
adjacencyAndFlankingCheck bs (i,j) = [ d | d@(_,_) <- directions, isAdjacentToOpp d && flankOpp d]
    where
        b = board bs
```

```
        curPlayer = curr_player bs
        oppPlayer = if curPlayer == 1 then 2 else 1

        isAdjacentToOpp (di,dj) = inBounds (i+di,j+dj) && getBoardVal b (i+di,j+dj) == oppPlayer

        {-
        Because of short-circuiting nature of &&, flankOpp will only be called if isAdjacentToOpp
            is True
        for the given direction (di,dj). As such, inside flankOpp, we can simplify checking
            flanking condition
        to be scanning towards direction (di,dj) and returning True if we find a disc of current
            player before hitting
        an empty spot or going out of bounds. Hitting a disc of current player means flanking
            condition is met in the
        (di,dj) direction because opponent's disc(s) is sandwiched between the new position (i,j)
            current player is
        going to take and current player's existing disc on board.
        -}
        flankOpp (di,dj) = scan (i + di + di) (j + dj + dj)
            where
                scan x y =
                    if not (inBounds (x,y)) then False
                    else case getBoardVal b (x,y) of
                        val | val == curPlayer -> True
                        val | val == oppPlayer -> scan (x + di) (y + dj)
                        _ -> False

{-
Given the position of the to-be-placed disc of current player and BoardState, update the discs on
    board and change curr_player to opponent player
to return an up-to-date BoardState to keep the game going
-}
updateTurn :: Position -> BoardState -> BoardState
-- Place a disc at `pos` for `curPlayer` and flip any sandwiched opponent disks
updateTurn (i,j) bs = BoardState {
    board = updatedBoard,
    curr_player = oppPlayer
}
    where
        b = board bs
        curPlayer = curr_player bs
        oppPlayer = if curPlayer == 1 then 2 else 1
        -- Figure out which directions are valid for flipping
        flippableDirections = adjacencyAndFlankingCheck bs (i,j)
        -- Helper function to get positions of all opponent discs to be flipped in one direction
            (di,dj)
        flipInDirection (di,dj) = scan (i + di) (j + dj) []
            where
                scan x y acc =
                    if not (inBounds (x,y)) then []
                    else case getBoardVal b (x,y) of
                        val | val == oppPlayer -> scan (x + di) (y + dj) ((x,y):acc)
                        val | val == curPlayer -> acc
                        _ -> []
        allPositionsToFlip = concatMap flipInDirection flippableDirections
        -- Replace index (i,j) and allPositionsToFlip with curPlayer
        updatedBoard = updateBoardIndexes b ((i,j):allPositionsToFlip) (replicate (1 + length
            allPositionsToFlip) curPlayer)
```

```haskell
{-
Count number of discs of given player on board
-}
countDisc :: BoardState -> Int -> Int
countDisc bs player = sum [ length (filter (==player) row) | row <- board bs ]

{-
checkWinner, return winner of board (1, 2), or 0 if tie
-}

checkWinner :: BoardState -> Int
checkWinner bs
    | p1Count > p2Count = 1
    | p2Count > p1Count = 2
    | otherwise = 0
    where
        p1Count = countDisc bs 1
        p2Count = countDisc bs 2

{-
Given a BoardState, return heuristic score of board
-}
evaluateBoard :: BoardState -> Int
evaluateBoard bs = 2*(cornerHeuristic bs) + (mobilityHeuristic bs) + (discCountHeuristic bs)


{-
Given a BoardState, return higher scores if current player has corners
-}

cornerHeuristic :: BoardState -> Int
cornerHeuristic bs = c1 + c2 + c3 + c4 where
    b = board bs
    curPlayer = curr_player bs
    c1 = if getBoardVal b (0,0) == curPlayer then 2 else 0
    c2 = if getBoardVal b (0,7) == curPlayer then 2 else 0
    c3 = if getBoardVal b (7,0) == curPlayer then 2 else 0
    c4 = if getBoardVal b (7,7) == curPlayer then 2 else 0

{-
Given a BoardState, return higher scores if current player has more moves
-}
mobilityHeuristic :: BoardState -> Int
mobilityHeuristic bs = length pos where
    pos = getPossibleMoves bs

{-
Given a Board state, return higher score if current player has more discs on board by taking this
    move
-}
discCountHeuristic :: BoardState -> Int
discCountHeuristic bs = (playerCount - oppCount) `div` 3
    where
        curPlayer = curr_player bs
        oppPlayer = if curPlayer == 1 then 2 else 1
        playerCount = countDisc bs curPlayer
        oppCount = countDisc bs oppPlayer
```

```haskell
{-
miniMax algo
isMaxizingPlayer: True if current player is maximizing player, False if minimizing player
-}
miniMax :: BoardState -> Int -> Bool -> Int
miniMax bs remainingDepth isMaximizingPlayer
    | remainingDepth == 0 || null moves = evaluateBoard bs
    | isMaximizingPlayer = maximum [miniMax (updateTurn move bs) (remainingDepth-1) False | move
        <- moves]
    | otherwise        = minimum [miniMax (updateTurn move bs) (remainingDepth-1) True | move <-
        moves]
    where
        moves = getPossibleMoves bs
```

```haskell
{-
gameLoop logic for alternating between players
-}
gameLoop :: BoardState -> IO ()
gameLoop bs = do
    let possibleMoves = getPossibleMoves bs
    if null possibleMoves
        then do
            let oppPlayer = if curr_player bs == 1 then 2 else 1
                oppPossibleMoves = getPossibleMoves (BoardState { board = board bs, curr_player =
                    oppPlayer })
            if null oppPossibleMoves
                then do
                    let winner = checkWinner bs
                    if winner == 0
                    then putStrLn "Game over! It's a tie!"
                    else do
                        putStrLn $ "Game over! Winner is Player " ++ show winner
                        putStrLn $ "Final Board:"
                        printBoard (board bs)
                        putStrLn $ "Player 1 discs: " ++ show (countDisc bs 1)
                        putStrLn $ "Player 2 discs: " ++ show (countDisc bs 2)
                else do
                    putStrLn $ "Player " ++ show (curr_player bs) ++ " has no moves. Skipping turn."
                    gameLoop (BoardState { board = board bs, curr_player = oppPlayer })
    else do
        putStrLn $ "Possible moves for Player " ++ show (curr_player bs) ++ ": " ++ show
            possibleMoves
        move <- if curr_player bs == 1
                    then do
                        -- Pick a random move for player 1
                        gen <- newStdGen
                        let (randomIndex, _) = randomR (0, length possibleMoves - 1) gen
                        let move = possibleMoves !! randomIndex
                        putStrLn $ "Player 1 (Random) chooses move " ++ show move
                        return move
                    else do
                        putStrLn $ "Player 2 (MiniMax) chooses move "
                        let possibleMovesWithScores = [ (m, miniMax (updateTurn m bs) 3 True) | m <-
                            possibleMoves ]
                        putStrLn "Player 2 possible moves and scores:"
```

```
                    mapM_ (\(m,score) -> putStrLn $ "Move: " ++ show m ++ ", Score: " ++ show
                        score) possibleMovesWithScores
                    let move = fst $ maximumBy (\(_,score1) (_,score2) -> compare score1 score2)
                        possibleMovesWithScores

                    putStrLn $ "Player 2 (MiniMax) chooses move " ++ show move
                    return move
        putStrLn $ "Player " ++ show (curr_player bs) ++ " places disc at " ++ show move
        putStrLn "BEFORE MOVE:"
        printBoard (board bs)
        let newGameState = updateTurn move bs
        putStrLn "AFTER MOVE:"
        printBoard (board newGameState)
        gameLoop newGameState

main :: IO ()
main = do
    gameLoop initializeBoard
    putStrLn "Thanks for playing!"
```

## A.2 othello-minimax-parlist.hs

```
import System.IO()
import Data.Int()
import GHC.Base()

import System.Posix.Internals()
import Data.List (maximumBy)
import Data.Ord()
import Debug.Trace()
import System.Random(newStdGen, randomR)
import Control.Parallel.Strategies

{-
This file contains logic for parallelized minimax without alpha beta pruning using parList

Commands to compile and run this program:
stack install random

stack ghc --package random -- -Wall -O2 -threaded -rtsopts -o othello-minimax-parlist
    othello-minimax-parlist.hs
./othello-minimax-parlist +RTS -N2 -s -l
./threadscope othello-minimax-parlist.eventlog
-}


-- Game types
type Position = (Int, Int)

type Board = [[Int]]


data BoardState = BoardState {
    board      :: Board,
    curr_player :: Int
} deriving (Show)
```

```
rows :: Int
rows = 8

cols :: Int
cols = 8

gameBoard :: Board
gameBoard = [[0 | _ <- [1..cols]] | _ <- [1..rows]] :: [[Int]]


updateBoardIndex :: Board -> Position -> Int -> Board
updateBoardIndex board (i,j) val =
    case splitAt i board of
        (prevRows, currRow : afterRows) ->
            case splitAt j currRow of
                (prevElems, _ : afterElems) ->
                    prevRows ++ [prevElems ++ [val] ++ afterElems] ++ afterRows
                (_, _) -> board
        (_, _) -> board


updateBoardIndexes :: Board -> [Position] -> [Int] -> Board
updateBoardIndexes board (x:xs) (y:ys) = updateBoardIndexes newBoard xs ys where
    newBoard = updateBoardIndex board x y
updateBoardIndexes board _ _ = board


{-
Initialize the gameboard to start configuration and set current player to 1
-}
initializeBoard :: BoardState
initializeBoard = BoardState {
    board = updateBoardIndexes gameBoard [(3,3),(4,4),(4,3),(3,4)] [1,1,2,2],
    curr_player = 1
}

{-
Return the number at index (i,j) of the board
-}
getBoardVal :: Board -> Position -> Int
getBoardVal board (i,j) = (board !! i) !! j

{-
Defined 8-way directions for checking adjacency and flanking rules
-}
directions :: [Position]
directions = [(-1,-1), (-1,0), (-1,1), (0,-1), (0,1), (1,-1), (1,0), (1,1)]

{-
Check if a given position is within bounds of the board
-}
inBounds :: Position -> Bool
inBounds (x,y) = x >= 0 && x < rows && y >= 0 && y < cols

{-
Given a board state, return the possible next moves for the player
-}
getPossibleMoves :: BoardState -> [Position]
getPossibleMoves bs = [ (i,j) | i <- [0..rows-1],
```

```
                        j <- [0..cols-1],
                        getBoardVal (board bs) (i,j) == 0, -- spot is not taken yet
                        not (null (adjacencyAndFlankingCheck bs (i,j))) -- adjacency and
                            flanking rules are both met
                        ]

{-
Check if the adjacency and flanking conditions are both met for a given position
Return a list of directions where both conditions are met
If the list is non-empty, then the position is a valid move that satisfy both adjancy and
    flankinng rules
-}
adjacencyAndFlankingCheck :: BoardState -> Position -> [Position]
adjacencyAndFlankingCheck bs (i,j) = [ d | d@(_,_) <- directions, isAdjacentToOpp d && flankOpp d]
    where
        b = board bs
        curPlayer = curr_player bs
        oppPlayer = if curPlayer == 1 then 2 else 1

        isAdjacentToOpp (di,dj) = inBounds (i+di,j+dj) && getBoardVal b (i+di,j+dj) == oppPlayer

        {-
        Because of short-circuiting nature of &&, flankOpp will only be called if isAdjacentToOpp
            is True
        for the given direction (di,dj). As such, inside flankOpp, we can simplify checking
            flanking condition
        to be scanning towards direction (di,dj) and returning True if we find a disc of current
            player before hitting
        an empty spot or going out of bounds. Hitting a disc of current player means flanking
            condition is met in the
        (di,dj) direction because opponent's disc(s) is sandwiched between the new position (i,j)
            current player is
        going to take and current player's existing disc on board.
        -}
        flankOpp (di,dj) = scan (i + di + di) (j + dj + dj)
            where
                scan x y =
                    if not (inBounds (x,y)) then False
                    else case getBoardVal b (x,y) of
                        val | val == curPlayer -> True
                        val | val == oppPlayer -> scan (x + di) (y + dj)
                        _ -> False


{-
Given the position of the to-be-placed disc of current player and BoardState, update the discs on
    board and change curr_player to opponent player
to return an up-to-date BoardState to keep the game going
-}
updateTurn :: Position -> BoardState -> BoardState
-- Place a disc at `pos` for `curPlayer` and flip any sandwiched opponent disks
updateTurn (i,j) bs = BoardState {
    board = updatedBoard,
    curr_player = oppPlayer
}
    where
        b = board bs
        curPlayer = curr_player bs
        oppPlayer = if curPlayer == 1 then 2 else 1
```

```
        -- Figure out which directions are valid for flipping
        flippableDirections = adjacencyAndFlankingCheck bs (i,j)
        -- Helper function to get positions of all opponent discs to be flipped in one direction
            (di,dj)
        flipInDirection (di,dj) = scan (i + di) (j + dj) []
            where
                scan x y acc =
                    if not (inBounds (x,y)) then []
                    else case getBoardVal b (x,y) of
                        val | val == oppPlayer -> scan (x + di) (y + dj) ((x,y):acc)
                        val | val == curPlayer -> acc
                        _ -> []
        allPositionsToFlip = concatMap flipInDirection flippableDirections
        -- Replace index (i,j) and allPositionsToFlip with curPlayer
        updatedBoard = updateBoardIndexes b ((i,j):allPositionsToFlip) (replicate (1 + length
            allPositionsToFlip) curPlayer)

{-
Count number of discs of given player on board
-}
countDisc :: BoardState -> Int -> Int
countDisc bs player = sum [ length (filter (==player) row) | row <- board bs ]

{-
checkWinner, return winner of board (1, 2), or 0 if tie
-}

checkWinner :: BoardState -> Int
checkWinner bs
    | p1Count > p2Count = 1
    | p2Count > p1Count = 2
    | otherwise = 0
    where
        p1Count = countDisc bs 1
        p2Count = countDisc bs 2

{-
Given a BoardState, return heuristic score of board
-}
-- TODO
evaluateBoard :: BoardState -> Int
evaluateBoard bs = 2*(cornerHeuristic bs) + (mobilityHeuristic bs) + (discCountHeuristic bs)


{-
Given a BoardState, return higher scores if current player has corners
-}

cornerHeuristic :: BoardState -> Int
cornerHeuristic bs = c1 + c2 + c3 + c4 where
    b = board bs
    curPlayer = curr_player bs
    c1 = if getBoardVal b (0,0) == curPlayer then 2 else 0
    c2 = if getBoardVal b (0,7) == curPlayer then 2 else 0
    c3 = if getBoardVal b (7,0) == curPlayer then 2 else 0
    c4 = if getBoardVal b (7,7) == curPlayer then 2 else 0

{-
Given a BoardState, return higher scores if current player has more moves
```

```
-}
mobilityHeuristic :: BoardState -> Int
mobilityHeuristic bs = length pos where
    pos = getPossibleMoves bs

{-
Given a Board state, return higher score if current player has more discs on board by taking this
    move
-}
discCountHeuristic :: BoardState -> Int
discCountHeuristic bs = (playerCount - oppCount) `div` 3
    where
        curPlayer = curr_player bs
        oppPlayer = if curPlayer == 1 then 2 else 1
        playerCount = countDisc bs curPlayer
        oppCount = countDisc bs oppPlayer




{-
miniMax recursive method
-}

miniMax :: BoardState -> Int -> Bool -> Int
-- isMaxizingPlayer: True if current player is maximizing player, False if minimizing player
miniMax bs remainingDepth isMaximizingPlayer
    | remainingDepth == 0 || null moves = evaluateBoard bs
    | isMaximizingPlayer = maximum [miniMax (updateTurn move bs) (remainingDepth-1) False | move
        <- moves]
    | otherwise = minimum [miniMax (updateTurn move bs) (remainingDepth-1) True | move <- moves]
    where
        moves = getPossibleMoves bs


{-
No printing gameLoop for benchmarking
-}
noPrintGameLoop :: BoardState -> IO ()
noPrintGameLoop bs = do
    let possibleMoves = getPossibleMoves bs
    if null possibleMoves
        then do
            let oppPlayer = if curr_player bs == 1 then 2 else 1
                oppPossibleMoves = getPossibleMoves (BoardState { board = board bs, curr_player =
                    oppPlayer })
            if null oppPossibleMoves
                then do
                    let winner = checkWinner bs
                    if winner == 0
                    then putStrLn "Game over! It's a tie!"
                    else do
                        putStrLn $ "Game over! Winner is Player " ++ show winner
                        putStrLn $ "Player 1 discs: " ++ show (countDisc bs 1)
                        putStrLn $ "Player 2 discs: " ++ show (countDisc bs 2)
                else do
                    putStrLn $ "Player " ++ show (curr_player bs) ++ " has no moves. Skipping turn."
                    noPrintGameLoop (BoardState { board = board bs, curr_player = oppPlayer })
    else do
```

```
        move <- if curr_player bs == 1
                then do
                    -- Pick a random move for player 1
                    gen <- newStdGen
                    let (randomIndex, _) = randomR (0, length possibleMoves - 1) gen
                    let move = possibleMoves !! randomIndex
                    return move
                else do
                    let possibleMovesWithScores = [ (m, miniMax (updateTurn m bs) 3 True) | m <-
                        possibleMoves ] `using` parList rdeepseq
                    let move = fst $ maximumBy (\(_,score1) (_,score2) -> compare score1 score2)
                        possibleMovesWithScores
                    return move
        let newGameState = updateTurn move bs
        noPrintGameLoop newGameState

main :: IO ()
main = do
    noPrintGameLoop initializeBoard
```

## A.3 othello-minimax-parbuffer.hs

```
import System.IO()
import Data.Int()
import GHC.Base()

import System.Posix.Internals()
import Data.List (maximumBy)
import Data.Ord()
import Debug.Trace()
import System.Random(newStdGen, randomR)
import Control.Parallel.Strategies

{-
This file contains logic for parallelized minimax without alpha beta pruning using parBuffer

Commands to compile and run this program:
stack install random

stack ghc --package random -- -Wall -O2 -threaded -rtsopts -o othello-minimax-parbuffer
    othello-minimax-parbuffer.hs
./othello-minimax-parbuffer +RTS -N2 -s -l
./threadscope othello-minimax-parbuffer.eventlog
-}


-- Game types
type Position = (Int, Int)

type Board = [[Int]]


data BoardState = BoardState {
    board      :: Board,
    curr_player :: Int
} deriving (Show)
```

```haskell
rows :: Int
rows = 8

cols :: Int
cols = 8

gameBoard :: Board
gameBoard = [[0 | _ <- [1..cols]] | _ <- [1..rows]] :: [[Int]]


updateBoardIndex :: Board -> Position -> Int -> Board
updateBoardIndex board (i,j) val =
    case splitAt i board of
        (prevRows, currRow : afterRows) ->
            case splitAt j currRow of
                (prevElems, _ : afterElems) ->
                    prevRows ++ [prevElems ++ [val] ++ afterElems] ++ afterRows
                (_, _) -> board
        (_, _) -> board


updateBoardIndexes :: Board -> [Position] -> [Int] -> Board
updateBoardIndexes board (x:xs) (y:ys) = updateBoardIndexes newBoard xs ys where
    newBoard = updateBoardIndex board x y
updateBoardIndexes board _ _ = board


{-
Initialize the gameboard to start configuration and set current player to 1
-}
initializeBoard :: BoardState
initializeBoard = BoardState {
    board = updateBoardIndexes gameBoard [(3,3),(4,4),(4,3),(3,4)] [1,1,2,2],
    curr_player = 1
}

{-
Return the number at index (i,j) of the board
-}
getBoardVal :: Board -> Position -> Int
getBoardVal board (i,j) = (board !! i) !! j

{-
Defined 8-way directions for checking adjacency and flanking rules
-}
directions :: [Position]
directions = [(-1,-1), (-1,0), (-1,1), (0,-1), (0,1), (1,-1), (1,0), (1,1)]

{-
Check if a given position is within bounds of the board
-}
inBounds :: Position -> Bool
inBounds (x,y) = x >= 0 && x < rows && y >= 0 && y < cols

{-
Given a board state, return the possible next moves for the player
-}
getPossibleMoves :: BoardState -> [Position]
getPossibleMoves bs = [ (i,j) | i <- [0..rows-1],
```

```
                            j <- [0..cols-1],
                            getBoardVal (board bs) (i,j) == 0, -- spot is not taken yet
                            not (null (adjacencyAndFlankingCheck bs (i,j))) -- adjacency and
                                flanking rules are both met
                            ]

{-
Check if the adjacency and flanking conditions are both met for a given position
Return a list of directions where both conditions are met
If the list is non-empty, then the position is a valid move that satisfy both adjancy and
    flankinng rules
-}
adjacencyAndFlankingCheck :: BoardState -> Position -> [Position]
adjacencyAndFlankingCheck bs (i,j) = [ d | d@(_,_) <- directions, isAdjacentToOpp d && flankOpp d]
    where
        b = board bs
        curPlayer = curr_player bs
        oppPlayer = if curPlayer == 1 then 2 else 1

        isAdjacentToOpp (di,dj) = inBounds (i+di,j+dj) && getBoardVal b (i+di,j+dj) == oppPlayer

        {-
        Because of short-circuiting nature of &&, flankOpp will only be called if isAdjacentToOpp
            is True
        for the given direction (di,dj). As such, inside flankOpp, we can simplify checking
            flanking condition
        to be scanning towards direction (di,dj) and returning True if we find a disc of current
            player before hitting
        an empty spot or going out of bounds. Hitting a disc of current player means flanking
            condition is met in the
        (di,dj) direction because opponent's disc(s) is sandwiched between the new position (i,j)
            current player is
        going to take and current player's existing disc on board.
        -}
        flankOpp (di,dj) = scan (i + di + di) (j + dj + dj)
            where
                scan x y =
                    if not (inBounds (x,y)) then False
                    else case getBoardVal b (x,y) of
                        val | val == curPlayer -> True
                        val | val == oppPlayer -> scan (x + di) (y + dj)
                        _ -> False


{-
Given the position of the to-be-placed disc of current player and BoardState, update the discs on
    board and change curr_player to opponent player
to return an up-to-date BoardState to keep the game going
-}
updateTurn :: Position -> BoardState -> BoardState
-- Place a disc at `pos` for `curPlayer` and flip any sandwiched opponent disks
updateTurn (i,j) bs = BoardState {
    board = updatedBoard,
    curr_player = oppPlayer
}
    where
        b = board bs
        curPlayer = curr_player bs
        oppPlayer = if curPlayer == 1 then 2 else 1
```

```
        -- Figure out which directions are valid for flipping
        flippableDirections = adjacencyAndFlankingCheck bs (i,j)
        -- Helper function to get positions of all opponent discs to be flipped in one direction
            (di,dj)
        flipInDirection (di,dj) = scan (i + di) (j + dj) []
            where
                scan x y acc =
                    if not (inBounds (x,y)) then []
                    else case getBoardVal b (x,y) of
                        val | val == oppPlayer -> scan (x + di) (y + dj) ((x,y):acc)
                        val | val == curPlayer -> acc
                        _ -> []
        allPositionsToFlip = concatMap flipInDirection flippableDirections
        -- Replace index (i,j) and allPositionsToFlip with curPlayer
        updatedBoard = updateBoardIndexes b ((i,j):allPositionsToFlip) (replicate (1 + length
            allPositionsToFlip) curPlayer)

{-
Count number of discs of given player on board
-}
countDisc :: BoardState -> Int -> Int
countDisc bs player = sum [ length (filter (==player) row) | row <- board bs ]

{-
checkWinner, return winner of board (1, 2), or 0 if tie
-}

checkWinner :: BoardState -> Int
checkWinner bs
    | p1Count > p2Count = 1
    | p2Count > p1Count = 2
    | otherwise = 0
    where
        p1Count = countDisc bs 1
        p2Count = countDisc bs 2

{-
Given a BoardState, return heuristic score of board
-}
-- TODO
evaluateBoard :: BoardState -> Int
evaluateBoard bs = 2*(cornerHeuristic bs) + (mobilityHeuristic bs) + (discCountHeuristic bs)


{-
Given a BoardState, return higher scores if current player has corners
-}

cornerHeuristic :: BoardState -> Int
cornerHeuristic bs = c1 + c2 + c3 + c4 where
    b = board bs
    curPlayer = curr_player bs
    c1 = if getBoardVal b (0,0) == curPlayer then 2 else 0
    c2 = if getBoardVal b (0,7) == curPlayer then 2 else 0
    c3 = if getBoardVal b (7,0) == curPlayer then 2 else 0
    c4 = if getBoardVal b (7,7) == curPlayer then 2 else 0

{-
Given a BoardState, return higher scores if current player has more moves
```

```
-}
mobilityHeuristic :: BoardState -> Int
mobilityHeuristic bs = length pos where
    pos = getPossibleMoves bs

{-
Given a Board state, return higher score if current player has more discs on board by taking this
    move
-}
discCountHeuristic :: BoardState -> Int
discCountHeuristic bs = (playerCount - oppCount) `div` 3
    where
        curPlayer = curr_player bs
        oppPlayer = if curPlayer == 1 then 2 else 1
        playerCount = countDisc bs curPlayer
        oppCount = countDisc bs oppPlayer


miniMax :: BoardState -> Int -> Bool -> Int
-- isMaxizingPlayer: True if current player is maximizing player, False if minimizing player
miniMax bs remainingDepth isMaximizingPlayer
    | remainingDepth == 0 || null moves = evaluateBoard bs
    | isMaximizingPlayer = maximum [miniMax (updateTurn move bs) (remainingDepth-1) False | move
        <- moves]
    | otherwise = minimum [miniMax (updateTurn move bs) (remainingDepth-1) True | move <- moves]
    where
        moves = getPossibleMoves bs


{-
No printing gameLoop for benchmarking
-}
noPrintGameLoop :: BoardState -> IO ()
noPrintGameLoop bs = do
    let possibleMoves = getPossibleMoves bs
    if null possibleMoves
        then do
            let oppPlayer = if curr_player bs == 1 then 2 else 1
                oppPossibleMoves = getPossibleMoves (BoardState { board = board bs, curr_player =
                    oppPlayer })
            if null oppPossibleMoves
                then do
                    let winner = checkWinner bs
                    if winner == 0
                    then putStrLn "Game over! It's a tie!"
                    else do
                        putStrLn $ "Game over! Winner is Player " ++ show winner
                        putStrLn $ "Player 1 discs: " ++ show (countDisc bs 1)
                        putStrLn $ "Player 2 discs: " ++ show (countDisc bs 2)
                else do
                    putStrLn $ "Player " ++ show (curr_player bs) ++ " has no moves. Skipping turn."
                    noPrintGameLoop (BoardState { board = board bs, curr_player = oppPlayer })
    else do
        move <- if curr_player bs == 1
                then do
                    -- Pick a random move for player 1
                    gen <- newStdGen
                    let (randomIndex, _) = randomR (0, length possibleMoves - 1) gen
                    let move = possibleMoves !! randomIndex
```

```
                    return move
                else do
                    let possibleMovesWithScores = [ (m, miniMax (updateTurn m bs) 3 True) | m <-
                        possibleMoves ] `using` parBuffer 150 rdeepseq
                    let move = fst $ maximumBy (\(_,score1) (_,score2) -> compare score1 score2)
                        possibleMovesWithScores
                    return move
        let newGameState = updateTurn move bs
        noPrintGameLoop newGameState

main :: IO ()
main = do
    noPrintGameLoop initializeBoard
```

## A.4 othello-minimax-ab-parlist.hs

```
import System.IO()
import Data.Int()
import GHC.Base()

import System.Posix.Internals()
import Data.List (maximumBy)
import Data.Ord()
import Debug.Trace()
import System.Random(newStdGen, randomR)
import Control.Parallel.Strategies


{-
This file contains logic for parallelized minimax with alpha beta pruning using parList

Commands to compile and run this program:
stack install random

stack ghc --package random -- -Wall -O2 -threaded -rtsopts -o othello-minimax-ab-parlist
    othello-minimax-ab-parlist.hs
./othello-minimax-ab-parlist +RTS -N2 -s -l
./threadscope othello-minimax-ab-parlist.eventlog
-}


-- Game types
type Position = (Int, Int)

type Board = [[Int]]


data BoardState = BoardState {
    board      :: Board,
    curr_player :: Int
} deriving (Show)

rows :: Int
rows = 8

cols :: Int
cols = 8
```

```
gameBoard :: Board
gameBoard = [[0 | _ <- [1..cols]] | _ <- [1..rows]] :: [[Int]]


updateBoardIndex :: Board -> Position -> Int -> Board
updateBoardIndex board (i,j) val =
    case splitAt i board of
        (prevRows, currRow : afterRows) ->
            case splitAt j currRow of
                (prevElems, _ : afterElems) ->
                    prevRows ++ [prevElems ++ [val] ++ afterElems] ++ afterRows
                (_, _) -> board
        (_, _) -> board


updateBoardIndexes :: Board -> [Position] -> [Int] -> Board
updateBoardIndexes board (x:xs) (y:ys) = updateBoardIndexes newBoard xs ys where
    newBoard = updateBoardIndex board x y
updateBoardIndexes board _ _ = board


{-
Initialize the gameboard to start configuration and set current player to 1
-}
initializeBoard :: BoardState
initializeBoard = BoardState {
    board = updateBoardIndexes gameBoard [(3,3),(4,4),(4,3),(3,4)] [1,1,2,2],
    curr_player = 1
}


{-
Return the number at index (i,j) of the board
-}
getBoardVal :: Board -> Position -> Int
getBoardVal board (i,j) = (board !! i) !! j

{-
Defined 8-way directions for checking adjacency and flanking rules
-}
directions :: [Position]
directions = [(-1,-1), (-1,0), (-1,1), (0,-1), (0,1), (1,-1), (1,0), (1,1)]

{-
Check if a given position is within bounds of the board
-}
inBounds :: Position -> Bool
inBounds (x,y) = x >= 0 && x < rows && y >= 0 && y < cols

{-
Given a board state, return the possible next moves for the player
-}
getPossibleMoves :: BoardState -> [Position]
getPossibleMoves bs = [ (i,j) | i <- [0..rows-1],
                            j <- [0..cols-1],
                            getBoardVal (board bs) (i,j) == 0, -- spot is not taken yet
                            not (null (adjacencyAndFlankingCheck bs (i,j))) -- adjacency and
                                flanking rules are both met
```

```
                          ]

{-
Check if the adjacency and flanking conditions are both met for a given position
Return a list of directions where both conditions are met
If the list is non-empty, then the position is a valid move that satisfy both adjancy and
    flankinng rules
-}
adjacencyAndFlankingCheck :: BoardState -> Position -> [Position]
adjacencyAndFlankingCheck bs (i,j) = [ d | d@(_,_) <- directions, isAdjacentToOpp d && flankOpp d]
    where
        b = board bs
        curPlayer = curr_player bs
        oppPlayer = if curPlayer == 1 then 2 else 1

        isAdjacentToOpp (di,dj) = inBounds (i+di,j+dj) && getBoardVal b (i+di,j+dj) == oppPlayer

        {-
        Because of short-circuiting nature of &&, flankOpp will only be called if isAdjacentToOpp
            is True
        for the given direction (di,dj). As such, inside flankOpp, we can simplify checking
            flanking condition
        to be scanning towards direction (di,dj) and returning True if we find a disc of current
            player before hitting
        an empty spot or going out of bounds. Hitting a disc of current player means flanking
            condition is met in the
        (di,dj) direction because opponent's disc(s) is sandwiched between the new position (i,j)
            current player is
        going to take and current player's existing disc on board.
        -}
        flankOpp (di,dj) = scan (i + di + di) (j + dj + dj)
            where
                scan x y =
                    if not (inBounds (x,y)) then False
                    else case getBoardVal b (x,y) of
                        val | val == curPlayer -> True
                        val | val == oppPlayer -> scan (x + di) (y + dj)
                        _ -> False


{-
Given the position of the to-be-placed disc of current player and BoardState, update the discs on
    board and change curr_player to opponent player
to return an up-to-date BoardState to keep the game going
-}
updateTurn :: Position -> BoardState -> BoardState
-- Place a disc at `pos` for `curPlayer` and flip any sandwiched opponent disks
updateTurn (i,j) bs = BoardState {
    board = updatedBoard,
    curr_player = oppPlayer
}
    where
        b = board bs
        curPlayer = curr_player bs
        oppPlayer = if curPlayer == 1 then 2 else 1
        -- Figure out which directions are valid for flipping
        flippableDirections = adjacencyAndFlankingCheck bs (i,j)
        -- Helper function to get positions of all opponent discs to be flipped in one direction
            (di,dj)
```

```
        flipInDirection (di,dj) = scan (i + di) (j + dj) []
            where
                scan x y acc =
                    if not (inBounds (x,y)) then []
                    else case getBoardVal b (x,y) of
                        val | val == oppPlayer -> scan (x + di) (y + dj) ((x,y):acc)
                        val | val == curPlayer -> acc
                        _ -> []
        allPositionsToFlip = concatMap flipInDirection flippableDirections
        -- Replace index (i,j) and allPositionsToFlip with curPlayer
        updatedBoard = updateBoardIndexes b ((i,j):allPositionsToFlip) (replicate (1 + length
            allPositionsToFlip) curPlayer)

{-
Count number of discs of given player on board
-}
countDisc :: BoardState -> Int -> Int
countDisc bs player = sum [ length (filter (==player) row) | row <- board bs ]

{-
checkWinner, return winner of board (1, 2), or 0 if tie
-}

checkWinner :: BoardState -> Int
checkWinner bs
    | p1Count > p2Count = 1
    | p2Count > p1Count = 2
    | otherwise = 0
    where
        p1Count = countDisc bs 1
        p2Count = countDisc bs 2

{-
Given a BoardState, return heuristic score of board
-}
evaluateBoard :: BoardState -> Int
evaluateBoard bs = 2*(cornerHeuristic bs) + (mobilityHeuristic bs) + (discCountHeuristic bs)


{-
Given a BoardState, return higher scores if current player has corners
-}

cornerHeuristic :: BoardState -> Int
cornerHeuristic bs = c1 + c2 + c3 + c4 where
    b = board bs
    curPlayer = curr_player bs
    c1 = if getBoardVal b (0,0) == curPlayer then 2 else 0
    c2 = if getBoardVal b (0,7) == curPlayer then 2 else 0
    c3 = if getBoardVal b (7,0) == curPlayer then 2 else 0
    c4 = if getBoardVal b (7,7) == curPlayer then 2 else 0

{-
Given a BoardState, return higher scores if current player has more moves
-}
mobilityHeuristic :: BoardState -> Int
mobilityHeuristic bs = length pos where
    pos = getPossibleMoves bs
```

```haskell
{-
Given a Board state, return higher score if current player has more discs on board by taking this
    move
-}
discCountHeuristic :: BoardState -> Int
discCountHeuristic bs = (playerCount - oppCount) `div` 3
    where
        curPlayer = curr_player bs
        oppPlayer = if curPlayer == 1 then 2 else 1
        playerCount = countDisc bs curPlayer
        oppCount = countDisc bs oppPlayer



{-
miniMax with AlphaBeta, referenced from
https://www.reddit.com/r/haskell/comments/1nigy1n/stumped_on_alpha_beta_pruning_in_haskell/

-}
miniMaxAlphaBeta :: BoardState -> Int -> Bool -> Int -> Int -> Int
miniMaxAlphaBeta bs 0 _ _ _ = evaluateBoard bs

miniMaxAlphaBeta bs remainingDepth True alpha beta = go alpha 0 [updateTurn move bs | move <-
    getPossibleMoves bs]
  where
    go a v (s:ss) = let v' = max v (miniMaxAlphaBeta s (remainingDepth - 1) False a beta)
                    in  if v' >= beta then v'
                        else go (max a v') v' ss
    go _ v [] = v

miniMaxAlphaBeta bs remainingDepth False alpha beta = go beta 10000 [updateTurn move bs | move <-
    getPossibleMoves bs]
  where
    go b v (s:ss) = let v' = min v (miniMaxAlphaBeta s (remainingDepth - 1) True alpha b)
                    in  if v' <= alpha then v'
                        else go (min b v') v' ss
    go _ v [] = v


{-
No printing gameLoop
-}
noPrintGameLoop :: BoardState -> IO ()
noPrintGameLoop bs = do
    let possibleMoves = getPossibleMoves bs
    if null possibleMoves
        then do
            let oppPlayer = if curr_player bs == 1 then 2 else 1
                oppPossibleMoves = getPossibleMoves (BoardState { board = board bs, curr_player =
                    oppPlayer })
            if null oppPossibleMoves
                then do
                    let winner = checkWinner bs
                    if winner == 0
                    then putStrLn "Game over! It's a tie!"
                    else do
                        putStrLn $ "Game over! Winner is Player " ++ show winner
                        putStrLn $ "Player 1 discs: " ++ show (countDisc bs 1)
                        putStrLn $ "Player 2 discs: " ++ show (countDisc bs 2)
```

```
            else do
                putStrLn $ "Player " ++ show (curr_player bs) ++ " has no moves. Skipping turn."
                noPrintGameLoop (BoardState { board = board bs, curr_player = oppPlayer })
    else do
        move <- if curr_player bs == 1
                then do
                    -- Pick a random move for player 1
                    gen <- newStdGen
                    let (randomIndex, _) = randomR (0, length possibleMoves - 1) gen
                    let move = possibleMoves !! randomIndex
                    return move
                else do
                    let possibleMovesWithScores = [ (m, miniMaxAlphaBeta (updateTurn m bs) 3
                        True 0 100000) | m <- possibleMoves ] `using` parList rdeepseq
                    let move = fst $ maximumBy (\(_,score1) (_,score2) -> compare score1 score2)
                        possibleMovesWithScores
                    return move
        let newGameState = updateTurn move bs
        noPrintGameLoop newGameState

main :: IO ()
main = do
    noPrintGameLoop initializeBoard
```

## A.5 othello-minimax-ab-parbuffer.hs

```
import System.IO()
import Data.Int()
import GHC.Base()

import System.Posix.Internals()
import Data.List (maximumBy)
import Data.Ord()
import Debug.Trace()
import System.Random(newStdGen, randomR)
import Control.Parallel.Strategies


{-
This file contains logic for parallelized minimax with alpha beta pruning using parBuffer

Commands to compile and run this program:
stack install random

stack ghc --package random -- -Wall -O2 -threaded -rtsopts -o othello-minimax-ab-parbuffer
    othello-minimax-ab-parbuffer.hs
./othello-minimax-ab-parbuffer +RTS -N2 -s -l
./threadscope othello-minimax-ab-parbuffer.eventlog
-}


-- Game types
type Position = (Int, Int)

type Board = [[Int]]
```

```haskell
data BoardState = BoardState {
    board      :: Board,
    curr_player :: Int
} deriving (Show)

rows :: Int
rows = 8

cols :: Int
cols = 8

gameBoard :: Board
gameBoard = [[0 | _ <- [1..cols]] | _ <- [1..rows]] :: [[Int]]


updateBoardIndex :: Board -> Position -> Int -> Board
updateBoardIndex board (i,j) val =
    case splitAt i board of
        (prevRows, currRow : afterRows) ->
            case splitAt j currRow of
                (prevElems, _ : afterElems) ->
                    prevRows ++ [prevElems ++ [val] ++ afterElems] ++ afterRows
                (_, _) -> board
        (_, _) -> board


updateBoardIndexes :: Board -> [Position] -> [Int] -> Board
updateBoardIndexes board (x:xs) (y:ys) = updateBoardIndexes newBoard xs ys where
    newBoard = updateBoardIndex board x y
updateBoardIndexes board _ _ = board


{-
Initialize the gameboard to start configuration and set current player to 1
-}
initializeBoard :: BoardState
initializeBoard = BoardState {
    board = updateBoardIndexes gameBoard [(3,3),(4,4),(4,3),(3,4)] [1,1,2,2],
    curr_player = 1
}

{-
Printing for Board
Commented out because timed parallel calls will not print boards
-}
-- printBoard :: Board -> IO ()
-- printBoard board = mapM_ (putStrLn . unwords . map show) board


{-
Return the number at index (i,j) of the board
-}
getBoardVal :: Board -> Position -> Int
getBoardVal board (i,j) = (board !! i) !! j

{-
Defined 8-way directions for checking adjacency and flanking rules
-}
directions :: [Position]
```

```haskell
directions = [(-1,-1), (-1,0), (-1,1), (0,-1), (0,1), (1,-1), (1,0), (1,1)]

{-
Check if a given position is within bounds of the board
-}
inBounds :: Position -> Bool
inBounds (x,y) = x >= 0 && x < rows && y >= 0 && y < cols

{-
Given a board state, return the possible next moves for the player
-}
getPossibleMoves :: BoardState -> [Position]
getPossibleMoves bs = [ (i,j) | i <- [0..rows-1],
                            j <- [0..cols-1],
                            getBoardVal (board bs) (i,j) == 0, -- spot is not taken yet
                            not (null (adjacencyAndFlankingCheck bs (i,j))) -- adjacency and
                                flanking rules are both met
                            ]

{-
Check if the adjacency and flanking conditions are both met for a given position
Return a list of directions where both conditions are met
If the list is non-empty, then the position is a valid move that satisfy both adjancy and
    flankinng rules
-}
adjacencyAndFlankingCheck :: BoardState -> Position -> [Position]
adjacencyAndFlankingCheck bs (i,j) = [ d | d@(_,_) <- directions, isAdjacentToOpp d && flankOpp d]
    where
        b = board bs
        curPlayer = curr_player bs
        oppPlayer = if curPlayer == 1 then 2 else 1

        isAdjacentToOpp (di,dj) = inBounds (i+di,j+dj) && getBoardVal b (i+di,j+dj) == oppPlayer

        {-
        Because of short-circuiting nature of &&, flankOpp will only be called if isAdjacentToOpp
            is True
        for the given direction (di,dj). As such, inside flankOpp, we can simplify checking
            flanking condition
        to be scanning towards direction (di,dj) and returning True if we find a disc of current
            player before hitting
        an empty spot or going out of bounds. Hitting a disc of current player means flanking
            condition is met in the
        (di,dj) direction because opponent's disc(s) is sandwiched between the new position (i,j)
            current player is
        going to take and current player's existing disc on board.
        -}
        flankOpp (di,dj) = scan (i + di + di) (j + dj + dj)
            where
                scan x y =
                    if not (inBounds (x,y)) then False
                    else case getBoardVal b (x,y) of
                        val | val == curPlayer -> True
                        val | val == oppPlayer -> scan (x + di) (y + dj)
                        _ -> False


{-
```

```
Given the position of the to-be-placed disc of current player and BoardState, update the discs on
    board and change curr_player to opponent player
to return an up-to-date BoardState to keep the game going
-}
updateTurn :: Position -> BoardState -> BoardState
-- Place a disc at `pos` for `curPlayer` and flip any sandwiched opponent disks
updateTurn (i,j) bs = BoardState {
    board = updatedBoard,
    curr_player = oppPlayer
}
    where
        b = board bs
        curPlayer = curr_player bs
        oppPlayer = if curPlayer == 1 then 2 else 1
        -- Figure out which directions are valid for flipping
        flippableDirections = adjacencyAndFlankingCheck bs (i,j)
        -- Helper function to get positions of all opponent discs to be flipped in one direction
            (di,dj)
        flipInDirection (di,dj) = scan (i + di) (j + dj) []
            where
                scan x y acc =
                    if not (inBounds (x,y)) then []
                    else case getBoardVal b (x,y) of
                        val | val == oppPlayer -> scan (x + di) (y + dj) ((x,y):acc)
                        val | val == curPlayer -> acc
                        _ -> []
        allPositionsToFlip = concatMap flipInDirection flippableDirections
        -- Replace index (i,j) and allPositionsToFlip with curPlayer
        updatedBoard = updateBoardIndexes b ((i,j):allPositionsToFlip) (replicate (1 + length
            allPositionsToFlip) curPlayer)

{-
Count number of discs of given player on board
-}
countDisc :: BoardState -> Int -> Int
countDisc bs player = sum [ length (filter (==player) row) | row <- board bs ]

{-
checkWinner, return winner of board (1, 2), or 0 if tie
-}

checkWinner :: BoardState -> Int
checkWinner bs
    | p1Count > p2Count = 1
    | p2Count > p1Count = 2
    | otherwise = 0
    where
        p1Count = countDisc bs 1
        p2Count = countDisc bs 2

{-
Given a BoardState, return heuristic score of board
-}
evaluateBoard :: BoardState -> Int
evaluateBoard bs = 2*(cornerHeuristic bs) + (mobilityHeuristic bs) + (discCountHeuristic bs)


{-
Given a BoardState, return higher scores if current player has corners
```

```
-}

cornerHeuristic :: BoardState -> Int
cornerHeuristic bs = c1 + c2 + c3 + c4 where
    b = board bs
    curPlayer = curr_player bs
    c1 = if getBoardVal b (0,0) == curPlayer then 2 else 0
    c2 = if getBoardVal b (0,7) == curPlayer then 2 else 0
    c3 = if getBoardVal b (7,0) == curPlayer then 2 else 0
    c4 = if getBoardVal b (7,7) == curPlayer then 2 else 0

{-
Given a BoardState, return higher scores if current player has more moves
-}
mobilityHeuristic :: BoardState -> Int
mobilityHeuristic bs = length pos where
    pos = getPossibleMoves bs

{-
Given a Board state, return higher score if current player has more discs on board by taking this
    move
-}
discCountHeuristic :: BoardState -> Int
discCountHeuristic bs = (playerCount - oppCount) `div` 3
    where
        curPlayer = curr_player bs
        oppPlayer = if curPlayer == 1 then 2 else 1
        playerCount = countDisc bs curPlayer
        oppCount = countDisc bs oppPlayer



{-
miniMax with AlphaBeta, referenced from
https://www.reddit.com/r/haskell/comments/1nigy1n/stumped_on_alpha_beta_pruning_in_haskell/

-}
miniMaxAlphaBeta :: BoardState -> Int -> Bool -> Int -> Int -> Int
miniMaxAlphaBeta bs 0 _ _ _ = evaluateBoard bs

miniMaxAlphaBeta bs remainingDepth True alpha beta = go alpha 0 [updateTurn move bs | move <-
    getPossibleMoves bs]
  where
    go a v (s:ss) = let v' = max v (miniMaxAlphaBeta s (remainingDepth - 1) False a beta)
                    in  if v' >= beta then v'
                        else go (max a v') v' ss
    go _ v [] = v

miniMaxAlphaBeta bs remainingDepth False alpha beta = go beta 10000 [updateTurn move bs | move <-
    getPossibleMoves bs]
  where
    go b v (s:ss) = let v' = min v (miniMaxAlphaBeta s (remainingDepth - 1) True alpha b)
                    in  if v' <= alpha then v'
                        else go (min b v') v' ss
    go _ v [] = v


{-
No printing gameLoop for benchmarking
```

```
-}
noPrintGameLoop :: BoardState -> IO ()
noPrintGameLoop bs = do
    let possibleMoves = getPossibleMoves bs
    if null possibleMoves
        then do
            let oppPlayer = if curr_player bs == 1 then 2 else 1
                oppPossibleMoves = getPossibleMoves (BoardState { board = board bs, curr_player =
                    oppPlayer })
            if null oppPossibleMoves
                then do
                    let winner = checkWinner bs
                    if winner == 0
                    then putStrLn "Game over! It's a tie!"
                    else do
                        putStrLn $ "Game over! Winner is Player " ++ show winner
                        putStrLn $ "Player 1 discs: " ++ show (countDisc bs 1)
                        putStrLn $ "Player 2 discs: " ++ show (countDisc bs 2)
                else do
                    putStrLn $ "Player " ++ show (curr_player bs) ++ " has no moves. Skipping turn."
                    noPrintGameLoop (BoardState { board = board bs, curr_player = oppPlayer })
    else do
        move <- if curr_player bs == 1
                    then do
                        -- Pick a random move for player 1
                        gen <- newStdGen
                        let (randomIndex, _) = randomR (0, length possibleMoves - 1) gen
                        let move = possibleMoves !! randomIndex
                        return move
                    else do
                        let possibleMovesWithScores = [ (m, miniMaxAlphaBeta (updateTurn m bs) 3
                            True 0 100000) | m <- possibleMoves ] `using` parBuffer 150 rdeepseq
                        let move = fst $ maximumBy (\(_,score1) (_,score2) -> compare score1 score2)
                            possibleMovesWithScores
                        return move
        let newGameState = updateTurn move bs
        noPrintGameLoop newGameState

main :: IO ()
main = do
    noPrintGameLoop initializeBoard
```

# B Monte Carlo Tree Search Code Implementation

## B.1 othello-mcts-seq.hs

```
{-# OPTIONS_GHC -Wno-unrecognised-pragmas #-}
{-# HLINT ignore "Use isJust" #-}
import System.Random (StdGen, newStdGen, randomR, split)
import Data.List (maximumBy)
import Data.Ord (comparing)

{-
This file contains logic for sequential MCTS.
The main function runs a playthrough of an Othello game, and prints
information for each turn. (Board state, available moves, selected move)
```

```
Commands to compile and run this program:
stack install random
stack ghc --package random -- -Wall -O2 -o othello-mcts-seq othello-mcts-seq.hs
./othello-mcts-seq

-}

-- Game types
type Position = (Int, Int)

type Board = [[Int]] -- 0: empty, 1: player 1's disk, 2: player 2's disk

data BoardState = BoardState {
    board      :: Board,
    curr_player :: Int
} deriving (Show)

rows :: Int
rows = 8

cols :: Int
cols = 8

gameBoard :: Board
gameBoard = [[0 | _ <- [1..cols]] | _ <- [1..rows]] :: [[Int]]

updateBoardIndex :: Board -> Position -> Int -> Board
updateBoardIndex board (i,j) val =
    case splitAt i board of
        (prevRows, currRow : afterRows) ->
            case splitAt j currRow of
                (prevElems, _ : afterElems) ->
                    prevRows ++ [prevElems ++ [val] ++ afterElems] ++ afterRows
                (_, _) -> board
        (_, _) -> board

updateBoardIndexes :: Board -> [Position] -> [Int] -> Board
updateBoardIndexes board (x:xs) (y:ys) = updateBoardIndexes newBoard xs ys where
    newBoard = updateBoardIndex board x y
updateBoardIndexes board _ _ = board

{-
Initialize the gameboard to start configuration and set current player to 1
-}
initializeBoard :: BoardState
initializeBoard = BoardState {
    board = updateBoardIndexes gameBoard [(3,3),(4,4),(4,3),(3,4)] [1,1,2,2],
    curr_player = 1
}

{-
Printing for Board
-}
printBoard :: Board -> IO ()
printBoard board = mapM_ (putStrLn . unwords . map show) board


{-
```

```
Return the number at index (i,j) of the board
-}
getBoardVal :: Board -> Position -> Int
getBoardVal board (i,j) = (board !! i) !! j

{-
Defined 8-way directions for checking adjacency and flanking rules
-}
directions :: [Position]
directions = [(-1,-1), (-1,0), (-1,1), (0,-1), (0,1), (1,-1), (1,0), (1,1)]

{-
Check if a given position is within bounds of the board
-}
inBounds :: Position -> Bool
inBounds (x,y) = x >= 0 && x < rows && y >= 0 && y < cols

{-
Given a board state, return the possible next moves for the player
-}
getPossibleMoves :: BoardState -> [Position]
getPossibleMoves bs = [ (i,j) | i <- [0..rows-1],
                            j <- [0..cols-1],
                            getBoardVal (board bs) (i,j) == 0, -- spot is not taken yet
                            not (null (adjacencyAndFlankingCheck bs (i,j))) -- adjacency and
                                flanking rules are both met
                            ]

{-
Check if the adjacency and flanking conditions are both met for a given position
Return a list of directions where both conditions are met
If the list is non-empty, then the position is a valid move that satisfy both adjancy and
    flankinng rules
-}
adjacencyAndFlankingCheck :: BoardState -> Position -> [Position]
adjacencyAndFlankingCheck bs (i,j) = [ d | d <- directions, isAdjacentToOpp d && flankOpp d]
    where
        b = board bs
        curPlayer = curr_player bs
        oppPlayer = if curPlayer == 1 then 2 else 1

        isAdjacentToOpp (di,dj) = inBounds (i+di,j+dj) && getBoardVal b (i+di,j+dj) == oppPlayer

        {-
        Because of short-circuiting nature of &&, flankOpp will only be called if isAdjacentToOpp
            is True
        for the given direction (di,dj). As such, inside flankOpp, we can simplify checking
            flanking condition
        to be scanning towards direction (di,dj) and returning True if we find a disc of current
            player before hitting
        an empty spot or going out of bounds. Hitting a disc of current player means flanking
            condition is met in the
        (di,dj) direction because opponent's disc(s) is sandwiched between the new position (i,j)
            current player is
        going to take and current player's existing disc on board.
        -}
        flankOpp (di,dj) = scan (i + di + di) (j + dj + dj)
            where
                scan x y =
```

```
                if not (inBounds (x,y)) then False
                else case getBoardVal b (x,y) of
                    val | val == curPlayer -> True
                    val | val == oppPlayer -> scan (x + di) (y + dj)
                    _ -> False

{-
Given the position of the to-be-placed disc of current player and BoardState, update the discs on
    board and change curr_player to opponent player
to return an up-to-date BoardState to keep the game going
-}
updateTurn :: Position -> BoardState -> BoardState
-- Place a disc at `pos` for `curPlayer` and flip any sandwiched opponent disks
updateTurn (i,j) bs = BoardState {
    board = updatedBoard,
    curr_player = oppPlayer
}
    where
        b = board bs
        curPlayer = curr_player bs
        oppPlayer = if curPlayer == 1 then 2 else 1
        -- Figure out which directions are valid for flipping
        flippableDirections = adjacencyAndFlankingCheck bs (i,j)
        -- Helper function to get positions of all opponent discs to be flipped in one direction
            (di,dj)
        flipInDirection (di,dj) = scan (i + di) (j + dj) []
            where
                scan x y acc =
                    if not (inBounds (x,y)) then []
                    else case getBoardVal b (x,y) of
                        val | val == oppPlayer -> scan (x + di) (y + dj) ((x,y):acc)
                        val | val == curPlayer -> acc
                        _ -> []
        allPositionsToFlip = concatMap flipInDirection flippableDirections
        -- Replace index (i,j) and allPositionsToFlip with curPlayer
        updatedBoard = updateBoardIndexes b ((i,j):allPositionsToFlip) (replicate (1 + length
            allPositionsToFlip) curPlayer)

{-
Count number of discs of given player on board
-}
countDisc :: BoardState -> Int -> Int
countDisc bs player = sum [ length (filter (==player) row) | row <- board bs ]

{-
checkWinner, return winner of board (1, 2), or 0 if tie
-}

checkWinner :: BoardState -> Int
checkWinner bs
    | p1Count > p2Count = 1
    | p2Count > p1Count = 2
    | otherwise = 0
    where
        p1Count = countDisc bs 1
        p2Count = countDisc bs 2

-- MCTS related functions start here
```

```haskell
-- Tutorial on MCTS, where the following code snippets are based off:
    https://www.youtube.com/watch?v=UXW2yZndl7U&t=1s
{-
MCTS Node data structure
-}
data MCTSNode = MCTSNode {
    state      :: BoardState,
    children   :: [MCTSNode],
    n_i        :: Int,
    t          :: Double,
    m          :: Maybe Position -- The move that led to this state; Nothing for root node so we
        use Maybe
} deriving (Show)

{-
Calculate UCT value for a given MCTSNode
-}
uctValue :: MCTSNode -> Int -> Double
uctValue node parentN =
    if n_i node == 0 then 1/0 -- infinity
    else (t node / fromIntegral (n_i node)) + (sqrt 2) * sqrt (log (fromIntegral parentN) /
        fromIntegral (n_i node))

{-
Selection: select the best child to perform simulation based on UCT values
Also keeps track of path from root to selected node for backpropagation later
-}
selection :: MCTSNode -> [MCTSNode] -> (MCTSNode, [MCTSNode])
selection node path =
    if null (children node)
    then (node, path)
    else selection bestChild (path ++ [node])
    where
        bestChild = maximumBy (comparing (\child -> uctValue child (n_i node))) (children node)



{-
Expansion: expand the selected node
For each avalable action from current, add a new state to the children of the node / to the tree
-}
expansion :: MCTSNode -> [Position] -> MCTSNode
expansion node []   = node -- No possible moves to expand any further
expansion node (x:xs) = expansion (node { children = newNode : children node }) xs
    where
        bsNew = updateTurn x (state node)
        newNode = MCTSNode {
            state = bsNew,
            children = [],
            n_i = 0,
            t = 0.0,
            m = Just x
        }

{-
Simulation: simulate a random playout from the given node's state until terminal state is reached
-}
simulation :: MCTSNode -> StdGen -> Double
simulation node gen = simulateFromState (state node) gen
```

```
simulateFromState :: BoardState -> StdGen -> Double
simulateFromState bs g =
    let moves = getPossibleMoves bs
    in if null moves
        then evaluateBoardMTCS bs
        else
            let (randomIndex, g') = randomR (0, length moves - 1) g
                move = moves !! randomIndex
                newBs = updateTurn move bs
            in simulateFromState newBs g'

evaluateBoardMTCS :: BoardState -> Double
evaluateBoardMTCS bs = fromIntegral (p2Count - p1Count)
    where
        p1Count = countDisc bs 1
        p2Count = countDisc bs 2

{-
Backpropagation: propagate n_i and t values along the path from selected node back to root
Also update children field of the node along the way
[MCTSNode] is of the path from leaf to root
           It is of the form [a, a's parent, a's grandparent, ...]
           So we use the previous node to update the children field of the current node
MCTSNode: previously updated node (the node we will use to update children field of current node
    in the path)
Double: reward obtained from simulation
MCTSNode: return the last node updated (AKA root node updated)
-}
backpropagation :: [MCTSNode] -> MCTSNode -> Double -> MCTSNode
backpropagation [] updatedNode _ = updatedNode
backpropagation (parent:rest) updatedChild reward =
    let updatedParent = parent {
            children = map (\c -> if m c == m updatedChild then updatedChild else c) (children
                parent),
            n_i = n_i parent + 1,
            t = t parent + reward
        }
    in backpropagation rest updatedParent reward

{-
Run one iteration of MCTS:
1. selection
2. expansion
3. simulation
4. backpropagation
-}
runMCTSIteration :: MCTSNode -> StdGen -> MCTSNode
runMCTSIteration root gen =
    -- Stage 1. selection, select the best leaf, starting from the root
    let (selectedNode, path) = selection root [] -- selection returns leaf node in current MCTS
        tree
        possibleMoves = getPossibleMoves (state selectedNode)
    in  if null possibleMoves -- Case 1: Terminal node, skip expansion & simulation
        then
            let reward = evaluateBoardMTCS (state selectedNode)
                selectedNode' = selectedNode {
                    n_i = n_i selectedNode + 1,
                    t = t selectedNode + reward
```

```
                }
                updatedRoot = backpropagation (reverse path) selectedNode' reward -- Stage 4.
                    Backpropagation
            in updatedRoot
        -- Case 2: selected node not visited before, skip expansion
        else if n_i selectedNode == 0 && m selectedNode /= Nothing
            then
                let reward = simulation selectedNode (snd $ split gen) -- Stage 3. Simulation
                    selectedNode' = selectedNode {
                        n_i = n_i selectedNode + 1,
                        t = t selectedNode + reward
                    }
                    updatedRoot = backpropagation (reverse path) selectedNode' reward
                in updatedRoot
            else -- Case 3: need to expand selected node
                let expandedNode = expansion selectedNode possibleMoves -- Stage 2. Expand
                    selectedNode
                    (childToSimulate, _) = selection expandedNode []
                    reward = simulation childToSimulate (snd $ split gen) -- Stage 3. Simulation
                    childToSimulate' = childToSimulate {
                        n_i = n_i childToSimulate + 1,
                        t = t childToSimulate + reward
                    }
                    updatedRoot = backpropagation (expandedNode : reverse path) childToSimulate'
                        reward
                in updatedRoot

{-
Run MCTS for a given number of iterations starting from the given root node
MCTSNode: root of tree
Int: number of iterations to run
StdGen: random generator for simulations
return new MCTSNode root where the children have been added/updated according to MCTS iterations
-}
runMCTS :: MCTSNode -> Int -> [StdGen] -> MCTSNode
runMCTS root _ [] = root
runMCTS root 0 _ = root
runMCTS root n (g:gs) = runMCTS (runMCTSIteration root g) (n - 1) gs

-- MCTS related functions end here


{-
gameLoop logic for alternating between players
-- To illustrate that the MCTS is working, we make player 2 use miniMax to pick its moves
-- For now, player 1 will just pick a random vailable move
-- We should be able to see player 2 winning more often than player 1
-}
gameLoop :: BoardState -> IO ()
gameLoop bs = do
    let possibleMoves = getPossibleMoves bs
    if null possibleMoves
        then do
            let oppPlayer = if curr_player bs == 1 then 2 else 1
                oppPossibleMoves = getPossibleMoves (BoardState { board = board bs, curr_player =
                    oppPlayer })
            if null oppPossibleMoves
                then do
                    let winner = checkWinner bs
```

```
                if winner == 0
                then putStrLn "Game over! It's a tie!"
                else do
                    putStrLn $ "Game over! Winner is Player " ++ show winner
                    -- putStrLn $ "Final Board:"
                    -- printBoard (board bs)
                    putStrLn $ "Player 1 discs: " ++ show (countDisc bs 1)
                    putStrLn $ "Player 2 discs: " ++ show (countDisc bs 2)
            else do
                putStrLn $ "Player " ++ show (curr_player bs) ++ " has no moves. Skipping turn."
                gameLoop (BoardState { board = board bs, curr_player = oppPlayer })
    else do
        -- putStrLn $ "Possible moves for Player " ++ show (curr_player bs) ++ ": " ++ show
            possibleMoves
        move <- if curr_player bs == 1
                then do
                    -- Pick a random move for player 1
                    gen <- newStdGen
                    let (randomIndex, _) = randomR (0, length possibleMoves - 1) gen
                    let move = possibleMoves !! randomIndex
                    -- putStrLn $ "Player 1 (Random) chooses move " ++ show move
                    return move
                else do
                    -- putStrLn $ "Player 2 (MCTS) chooses move "
                    let rootNode = MCTSNode {
                        state = bs,
                        children = [],
                        n_i = 0,
                        t = 0.0,
                        m = Nothing
                    }
                    gen <- newStdGen
                    let gens = take 1000 $ iterate (snd . split) gen
                        mctsRoot = runMCTS rootNode 1000 gens -- Run MCTS for 1000 iterations
                    -- Print statements to make sure the MTCS actually works and results are
                        propagated back to root
                    putStrLn $ "Possible moves for Player 2 " ++ show (curr_player bs) ++ ": "
                        ++ show possibleMoves
                    putStrLn $ "MCTS Root after iterations: " ++ show mctsRoot
                    let bestChild = maximumBy (comparing (\child -> uctValue child (n_i
                        mctsRoot))) (children mctsRoot)
                    let move = case m bestChild of
                                Just pos -> pos
                                Nothing -> error "No move found. Something went wrong with
                                    MCTS implementation."
                    putStrLn $ "Player 2 (MCTS) chooses move " ++ show move
                    return move
        putStrLn $ "Player " ++ show (curr_player bs) ++ " places disc at " ++ show move
        putStrLn "BEFORE MOVE:"
        printBoard (board bs)
        let newGameState = updateTurn move bs
        putStrLn "AFTER MOVE:"
        printBoard (board newGameState)
        gameLoop newGameState


main :: IO ()
main = do
    gameLoop initializeBoard
```

```
    putStrLn "Thanks for playing!"
```

## B.2 othello-mcts-par-v1.hs

```
{-# LANGUAGE DeriveAnyClass #-}
{-# LANGUAGE DeriveGeneric #-}
{-# OPTIONS_GHC -Wno-unrecognised-pragmas #-}
{-# HLINT ignore "Avoid lambda" #-}
import System.Random (StdGen, newStdGen, randomR, split)
import Data.List (maximumBy)
import Data.List.Split (chunksOf)
import Data.Ord (comparing)
import Control.Parallel.Strategies
import GHC.Generics (Generic)

{-
This file contains logic for root parallelized MCTS (version 1)
If replicating result from Attempt 1, replace last number on line
"let mctsRoot = runMCTSParallel rootNode 1000 gen 1" with number of threads to parallelize on

Commands to compile and run this program:
stack install random
stack install split

stack ghc --package random -- -Wall -O2 -threaded -rtsopts -o othello-mcts-par-v1
    othello-mcts-par-v1.hs
./othello-mcts-par-v1 +RTS -N2 -s -l
./threadscope othello-mcts-par-v1.eventlog
-}

-- Game types
type Position = (Int, Int)

type Board = [[Int]] -- 0: empty, 1: player 1's disk, 2: player 2's disk

data BoardState = BoardState {
    board      :: Board,
    curr_player :: Int
} deriving (Show, Generic, NFData)

rows :: Int
rows = 8

cols :: Int
cols = 8

gameBoard :: Board
gameBoard = [[0 | _ <- [1..cols]] | _ <- [1..rows]] :: [[Int]]

updateBoardIndex :: Board -> Position -> Int -> Board
updateBoardIndex board (i,j) val =
    case splitAt i board of
        (prevRows, currRow : afterRows) ->
            case splitAt j currRow of
                (prevElems, _ : afterElems) ->
                    prevRows ++ [prevElems ++ [val] ++ afterElems] ++ afterRows
                (_, _) -> board
```

```
        (_, _) -> board

updateBoardIndexes :: Board -> [Position] -> [Int] -> Board
updateBoardIndexes board (x:xs) (y:ys) = updateBoardIndexes newBoard xs ys where
    newBoard = updateBoardIndex board x y
updateBoardIndexes board _ _ = board

{-
Initialize the gameboard to start configuration and set current player to 1
-}
initializeBoard :: BoardState
initializeBoard = BoardState {
    board = updateBoardIndexes gameBoard [(3,3),(4,4),(4,3),(3,4)] [1,1,2,2],
    curr_player = 1
}

{-
Return the number at index (i,j) of the board
-}
getBoardVal :: Board -> Position -> Int
getBoardVal board (i,j) = (board !! i) !! j

{-
Defined 8-way directions for checking adjacency and flanking rules
-}
directions :: [Position]
directions = [(-1,-1), (-1,0), (-1,1), (0,-1), (0,1), (1,-1), (1,0), (1,1)]

{-
Check if a given position is within bounds of the board
-}
inBounds :: Position -> Bool
inBounds (x,y) = x >= 0 && x < rows && y >= 0 && y < cols

{-
Given a board state, return the possible next moves for the player
-}
getPossibleMoves :: BoardState -> [Position]
getPossibleMoves bs = [ (i,j) | i <- [0..rows-1],
                            j <- [0..cols-1],
                            getBoardVal (board bs) (i,j) == 0, -- spot is not taken yet
                            not (null (adjacencyAndFlankingCheck bs (i,j))) -- adjacency and
                                flanking rules are both met
                            ]

{-
Check if the adjacency and flanking conditions are both met for a given position
Return a list of directions where both conditions are met
If the list is non-empty, then the position is a valid move that satisfy both adjancy and
    flankinng rules
-}
adjacencyAndFlankingCheck :: BoardState -> Position -> [Position]
adjacencyAndFlankingCheck bs (i,j) = [ d | d <- directions, isAdjacentToOpp d && flankOpp d]
    where
        b = board bs
        curPlayer = curr_player bs
        oppPlayer = if curPlayer == 1 then 2 else 1

        isAdjacentToOpp (di,dj) = inBounds (i+di,j+dj) && getBoardVal b (i+di,j+dj) == oppPlayer
```

```
        {-
        Because of short-circuiting nature of &&, flankOpp will only be called if isAdjacentToOpp
            is True
        for the given direction (di,dj). As such, inside flankOpp, we can simplify checking
            flanking condition
        to be scanning towards direction (di,dj) and returning True if we find a disc of current
            player before hitting
        an empty spot or going out of bounds. Hitting a disc of current player means flanking
            condition is met in the
        (di,dj) direction because opponent's disc(s) is sandwiched between the new position (i,j)
            current player is
        going to take and current player's existing disc on board.
        -}
        flankOpp (di,dj) = scan (i + di + di) (j + dj + dj)
            where
                scan x y =
                    if not (inBounds (x,y)) then False
                    else case getBoardVal b (x,y) of
                        val | val == curPlayer -> True
                        val | val == oppPlayer -> scan (x + di) (y + dj)
                        _ -> False

{-
Given the position of the to-be-placed disc of current player and BoardState, update the discs on
    board and change curr_player to opponent player
to return an up-to-date BoardState to keep the game going
-}
updateTurn :: Position -> BoardState -> BoardState
-- Place a disc at `pos` for `curPlayer` and flip any sandwiched opponent disks
updateTurn (i,j) bs = BoardState {
    board = updatedBoard,
    curr_player = oppPlayer
}
    where
        b = board bs
        curPlayer = curr_player bs
        oppPlayer = if curPlayer == 1 then 2 else 1
        -- Figure out which directions are valid for flipping
        flippableDirections = adjacencyAndFlankingCheck bs (i,j)
        -- Helper function to get positions of all opponent discs to be flipped in one direction
            (di,dj)
        flipInDirection (di,dj) = scan (i + di) (j + dj) []
            where
                scan x y acc =
                    if not (inBounds (x,y)) then []
                    else case getBoardVal b (x,y) of
                        val | val == oppPlayer -> scan (x + di) (y + dj) ((x,y):acc)
                        val | val == curPlayer -> acc
                        _ -> []
        allPositionsToFlip = concatMap flipInDirection flippableDirections
        -- Replace index (i,j) and allPositionsToFlip with curPlayer
        updatedBoard = updateBoardIndexes b ((i,j):allPositionsToFlip) (replicate (1 + length
            allPositionsToFlip) curPlayer)

{-
Count number of discs of given player on board
-}
countDisc :: BoardState -> Int -> Int
```

```haskell
countDisc bs player = sum [ length (filter (==player) row) | row <- board bs ]

{-
checkWinner, return winner of board (1, 2), or 0 if tie
-}

checkWinner :: BoardState -> Int
checkWinner bs
    | p1Count > p2Count = 1
    | p2Count > p1Count = 2
    | otherwise = 0
    where
        p1Count = countDisc bs 1
        p2Count = countDisc bs 2

-- MCTS related functions start here
-- Tutorial on MCTS, where the following code snippets are based off:
    https://www.youtube.com/watch?v=UXW2yZndl7U&t=1s
{-
MCTS Node data structure
-}
data MCTSNode = MCTSNode {
    state      :: BoardState,
    children   :: [MCTSNode],
    n_i        :: Int,
    t          :: Double,
    m          :: Maybe Position -- The move that led to this state; Nothing for root node so we
        use Maybe
} deriving (Show, Generic, NFData)

{-
Calculate UCT value for a given MCTSNode
-}
uctValue :: MCTSNode -> Int -> Double
uctValue node parentN =
    if n_i node == 0 then 1/0 -- infinity
    else (t node / fromIntegral (n_i node)) + (sqrt 2) * sqrt (log (fromIntegral parentN) /
        fromIntegral (n_i node))

{-
Selection: select the best child to perform simulation based on UCT values
Also keeps track of path from root to selected node for backpropagation later
-}
selection :: MCTSNode -> [MCTSNode] -> (MCTSNode, [MCTSNode])
selection node path =
    if null (children node)
    then (node, path)
    else selection bestChild (path ++ [node])
    where
        bestChild = maximumBy (comparing (\child -> uctValue child (n_i node))) (children node)



{-
Expansion: expand the selected node
For each avalable action from current, add a new state to the children of the node / to the tree
-}
expansion :: MCTSNode -> [Position] -> MCTSNode
expansion node []   = node -- No possible moves to expand any further
```

```
expansion node (x:xs) = expansion (node { children = newNode : children node }) xs
    where
        bsNew = updateTurn x (state node)
        newNode = MCTSNode {
            state = bsNew,
            children = [],
            n_i = 0,
            t = 0.0,
            m = Just x
        }

{-
Simulation: simulate a random playout from the given node's state until terminal state is reached
-}
simulation :: MCTSNode -> StdGen -> Double
simulation node gen = simulateFromState (state node) gen

simulateFromState :: BoardState -> StdGen -> Double
simulateFromState bs g =
    let moves = getPossibleMoves bs
    in if null moves
        then evaluateBoardMTCS bs
        else
            let (randomIndex, g') = randomR (0, length moves - 1) g
                move = moves !! randomIndex
                newBs = updateTurn move bs
            in simulateFromState newBs g'

evaluateBoardMTCS :: BoardState -> Double
evaluateBoardMTCS bs = fromIntegral (p2Count - p1Count)
    where
        p1Count = countDisc bs 1
        p2Count = countDisc bs 2

{-
Backpropagation: propagate n_i and t values along the path from selected node back to root
Also update children field of the node along the way
[MCTSNode] is of the path from leaf to root
           It is of the form [a, a's parent, a's grandparent, ...]
           So we use the previous node to update the children field of the current node
MCTSNode: previously updated node (the node we will use to update children field of current node
    in the path)
Double: reward obtained from simulation
MCTSNode: return the last node updated (AKA root node updated)
-}
backpropagation :: [MCTSNode] -> MCTSNode -> Double -> MCTSNode
backpropagation [] updatedNode _ = updatedNode
backpropagation (parent:rest) updatedChild reward =
    let updatedParent = parent {
            children = map (\c -> if m c == m updatedChild then updatedChild else c) (children
                parent),
            n_i = n_i parent + 1,
            t = t parent + reward
        }
    in backpropagation rest updatedParent reward

{-
Run one iteration of MCTS:
1. selection
```

```
2. expansion
3. simulation
4. backpropagation
-}
runMCTSIteration :: MCTSNode -> StdGen -> MCTSNode
runMCTSIteration root gen =
    -- 1. selection, select the best leaf, starting from the root
    let (selectedNode, path) = selection root [] -- selection returns leaf node in current MCTS
         tree
        possibleMoves = getPossibleMoves (state selectedNode)
    in  if null possibleMoves
        then
            let reward = evaluateBoardMTCS (state selectedNode) -- Terminal node: can directly
                 evaluate without simulation
                selectedNode' = selectedNode {
                    n_i = n_i selectedNode + 1,
                    t = t selectedNode + reward
                }
                updatedRoot = backpropagation (reverse path) selectedNode' reward -- Still need to
                     backpropagate the reward up the tree
            in updatedRoot
        else if n_i selectedNode == 0 && m selectedNode /= Nothing -- is n_i for current 0? Also
             need to make sure it's not root
            then
                -- Directly simulate/rollout from selectedNode without expansion
                let reward = simulation selectedNode (snd $ split gen) -- 3. Simulation
                    selectedNode' = selectedNode {
                        n_i = n_i selectedNode + 1,
                        t = t selectedNode + reward
                    }
                    updatedRoot = backpropagation (reverse path) selectedNode' reward
                in updatedRoot
            else
                -- Expand selectedNode, then simulate/rollout from the new child
                let expandedNode = expansion selectedNode possibleMoves
                    (childToSimulate, _) = selection expandedNode []
                    reward = simulation childToSimulate (snd $ split gen)
                    -- expandedNode is the parent of the updated child (childToSimulate')
                    childToSimulate' = childToSimulate {
                        n_i = n_i childToSimulate + 1,
                        t = t childToSimulate + reward
                    }
                    updatedRoot = backpropagation (expandedNode : reverse path) childToSimulate'
                         reward
                in updatedRoot

{-
Run MCTS for a given number of iterations starting from the given root node
MCTSNode: root of tree
Int: number of iterations to run
StdGen: random generator for simulations
return new MCTSNode root where the children have been added/updated according to MCTS iterations
-}
runMCTS :: MCTSNode -> Int -> [StdGen] -> MCTSNode
runMCTS root _ [] = root
runMCTS root 0 _ = root
runMCTS root n (g:gs) = runMCTS (runMCTSIteration root g) (n - 1) gs

{-
```

```
Version 1: split iterations evenly among threads
For simplicity, if iterations is not divisible by threadNum, we just give every thread an extra
    iteration
instead of figuring out how to distribute the remainder r (which may incur more overhead for
    distribution)
This also simplifies the code for splitting the random generators and for calling parMap rdeepseq
-}
runMCTSParallel :: MCTSNode -> Int -> StdGen -> Int -> MCTSNode
runMCTSParallel root iterations gen threadNum = mergedRoot
    where
        (q, r) = iterations `divMod` threadNum
        iterationsPerThread = if r == 0 then q else q + 1
        gens = take (iterationsPerThread * threadNum) $ iterate (snd . split) gen
        gensGrouped = chunksOf iterationsPerThread gens
        updatedRoots = parMap rdeepseq (\g -> runMCTS root iterationsPerThread g) gensGrouped
        mergedRoot = mergeMCTSRuns (tail updatedRoots) (head updatedRoots)

mergeMCTSRuns :: [MCTSNode] -> MCTSNode -> MCTSNode
mergeMCTSRuns [] root = root
mergeMCTSRuns (x:xs) root = mergeMCTSRuns xs mergedRoot
    where
        mergedRoot = root {
            n_i = n_i root + n_i x,
            t = t root + t x,
            children = mergeChildren (children root) (children x)
        }

mergeChildren :: [MCTSNode] -> [MCTSNode] -> [MCTSNode]
mergeChildren (x:xs) ys = mergedChild : mergeChildren xs ys
    where
        move = m x
        y = head $ filter (\child -> m child == move) ys -- find corresponding child in ys based on
            move
        mergedChild = x {
            n_i = n_i x + n_i y,
            t = t x + t y,
            m = move, -- both children should have same move
            children = [] -- We don't need to go deeper since we only care about root's children
                for MCTS decision making
        }
mergeChildren _ _ = [] -- children root and children x should be the same, except each child's n_i
    and t
                    -- so both lists should have same length, and we don't need to worry about
                        mismatch in length
-- MCTS related functions end here


{-
noPrintGameLoop logic for alternating between players without printing boards
-}
noPrintGameLoop :: BoardState -> IO ()
noPrintGameLoop bs = do
    let possibleMoves = getPossibleMoves bs
    if null possibleMoves
        then do
            let oppPlayer = if curr_player bs == 1 then 2 else 1
                oppPossibleMoves = getPossibleMoves (BoardState { board = board bs, curr_player =
                    oppPlayer })
            if null oppPossibleMoves
```

```
            then do
                let winner = checkWinner bs
                if winner == 0
                then putStrLn "Game over! It's a tie!"
                else do
                    putStrLn $ "Game over! Winner is Player " ++ show winner
                    putStrLn $ "Player 1 discs: " ++ show (countDisc bs 1)
                    putStrLn $ "Player 2 discs: " ++ show (countDisc bs 2)
            else do
                putStrLn $ "Player " ++ show (curr_player bs) ++ " has no moves. Skipping turn."
                noPrintGameLoop (BoardState { board = board bs, curr_player = oppPlayer })
    else do
        move <- if curr_player bs == 1
                then do
                    -- Pick a random move for player 1
                    gen <- newStdGen
                    let (randomIndex, _) = randomR (0, length possibleMoves - 1) gen
                    let move = possibleMoves !! randomIndex
                    return move
                else do
                    let rootNode = MCTSNode {
                        state = bs,
                        children = [],
                        n_i = 0,
                        t = 0.0,
                        m = Nothing
                    }
                    gen <- newStdGen
                    -- Version 1: Root Parallelization, run MCTS for 1000 iterations in parallel
                    let mctsRoot = runMCTSParallel rootNode 1000 gen 8 -- Change the last number
                        to indicate number of threads we can parallelize on
                    let bestChild = maximumBy (comparing (\child -> uctValue child (n_i
                        mctsRoot))) (children mctsRoot)
                        move = case m bestChild of
                                    Just pos -> pos
                                    Nothing -> error "No move found. Something went wrong with
                                        MCTS implementation."
                    return move
        let newGameState = updateTurn move bs
        noPrintGameLoop newGameState

main :: IO ()
main = do
    noPrintGameLoop initializeBoard
```

## B.3 othello-mcts-par-v2.hs

```
{-# LANGUAGE DeriveAnyClass #-}
{-# LANGUAGE DeriveGeneric #-}
{-# OPTIONS_GHC -Wno-unrecognised-pragmas #-}
{-# HLINT ignore "Avoid lambda" #-}
import System.Random (StdGen, newStdGen, randomR, split)
import Data.List (maximumBy)
import Data.List.Split (chunksOf)
import Data.Ord (comparing)
import Control.Parallel.Strategies
import GHC.Generics (Generic)
```

```haskell
{-
This file contains logic for chunked root parallelized MCTS (version 2)

Commands to compile and run this program:
stack install random
stack install split

stack ghc --package random -- -Wall -O2 -threaded -rtsopts -o othello-mcts-par-v2
    othello-mcts-par-v2.hs
./othello-mcts-par-v2 +RTS -N2 -s -l
./threadscope othello-mcts-par-v2.eventlog
-}

-- Game types
type Position = (Int, Int)

type Board = [[Int]] -- 0: empty, 1: player 1's disk, 2: player 2's disk

data BoardState = BoardState {
    board      :: Board,
    curr_player :: Int
} deriving (Show, Generic, NFData)

rows :: Int
rows = 8

cols :: Int
cols = 8

gameBoard :: Board
gameBoard = [[0 | _ <- [1..cols]] | _ <- [1..rows]] :: [[Int]]

updateBoardIndex :: Board -> Position -> Int -> Board
updateBoardIndex board (i,j) val =
    case splitAt i board of
        (prevRows, currRow : afterRows) ->
            case splitAt j currRow of
                (prevElems, _ : afterElems) ->
                    prevRows ++ [prevElems ++ [val] ++ afterElems] ++ afterRows
                (_, _) -> board
        (_, _) -> board

updateBoardIndexes :: Board -> [Position] -> [Int] -> Board
updateBoardIndexes board (x:xs) (y:ys) = updateBoardIndexes newBoard xs ys where
    newBoard = updateBoardIndex board x y
updateBoardIndexes board _ _ = board

{-
Initialize the gameboard to start configuration and set current player to 1
-}
initializeBoard :: BoardState
initializeBoard = BoardState {
    board = updateBoardIndexes gameBoard [(3,3),(4,4),(4,3),(3,4)] [1,1,2,2],
    curr_player = 1
}

{-
Return the number at index (i,j) of the board
```

```
-}
getBoardVal :: Board -> Position -> Int
getBoardVal board (i,j) = (board !! i) !! j

{-
Defined 8-way directions for checking adjacency and flanking rules
-}
directions :: [Position]
directions = [(-1,-1), (-1,0), (-1,1), (0,-1), (0,1), (1,-1), (1,0), (1,1)]

{-
Check if a given position is within bounds of the board
-}
inBounds :: Position -> Bool
inBounds (x,y) = x >= 0 && x < rows && y >= 0 && y < cols

{-
Given a board state, return the possible next moves for the player
-}
getPossibleMoves :: BoardState -> [Position]
getPossibleMoves bs = [ (i,j) | i <- [0..rows-1],
                            j <- [0..cols-1],
                            getBoardVal (board bs) (i,j) == 0, -- spot is not taken yet
                            not (null (adjacencyAndFlankingCheck bs (i,j))) -- adjacency and
                                flanking rules are both met
                            ]

{-
Check if the adjacency and flanking conditions are both met for a given position
Return a list of directions where both conditions are met
If the list is non-empty, then the position is a valid move that satisfy both adjancy and
    flankinng rules
-}
adjacencyAndFlankingCheck :: BoardState -> Position -> [Position]
adjacencyAndFlankingCheck bs (i,j) = [ d | d <- directions, isAdjacentToOpp d && flankOpp d]
    where
        b = board bs
        curPlayer = curr_player bs
        oppPlayer = if curPlayer == 1 then 2 else 1

        isAdjacentToOpp (di,dj) = inBounds (i+di,j+dj) && getBoardVal b (i+di,j+dj) == oppPlayer

        {-
        Because of short-circuiting nature of &&, flankOpp will only be called if isAdjacentToOpp
            is True
        for the given direction (di,dj). As such, inside flankOpp, we can simplify checking
            flanking condition
        to be scanning towards direction (di,dj) and returning True if we find a disc of current
            player before hitting
        an empty spot or going out of bounds. Hitting a disc of current player means flanking
            condition is met in the
        (di,dj) direction because opponent's disc(s) is sandwiched between the new position (i,j)
            current player is
        going to take and current player's existing disc on board.
        -}
        flankOpp (di,dj) = scan (i + di + di) (j + dj + dj)
            where
                scan x y =
                    if not (inBounds (x,y)) then False
```

```
                else case getBoardVal b (x,y) of
                    val | val == curPlayer -> True
                    val | val == oppPlayer -> scan (x + di) (y + dj)
                    _ -> False

{-
Given the position of the to-be-placed disc of current player and BoardState, update the discs on
    board and change curr_player to opponent player
to return an up-to-date BoardState to keep the game going
-}
updateTurn :: Position -> BoardState -> BoardState
-- Place a disc at `pos` for `curPlayer` and flip any sandwiched opponent disks
updateTurn (i,j) bs = BoardState {
    board = updatedBoard,
    curr_player = oppPlayer
}
    where
        b = board bs
        curPlayer = curr_player bs
        oppPlayer = if curPlayer == 1 then 2 else 1
        -- Figure out which directions are valid for flipping
        flippableDirections = adjacencyAndFlankingCheck bs (i,j)
        -- Helper function to get positions of all opponent discs to be flipped in one direction
            (di,dj)
        flipInDirection (di,dj) = scan (i + di) (j + dj) []
            where
                scan x y acc =
                    if not (inBounds (x,y)) then []
                    else case getBoardVal b (x,y) of
                        val | val == oppPlayer -> scan (x + di) (y + dj) ((x,y):acc)
                        val | val == curPlayer -> acc
                        _ -> []
        allPositionsToFlip = concatMap flipInDirection flippableDirections
        -- Replace index (i,j) and allPositionsToFlip with curPlayer
        updatedBoard = updateBoardIndexes b ((i,j):allPositionsToFlip) (replicate (1 + length
            allPositionsToFlip) curPlayer)

{-
Count number of discs of given player on board
-}
countDisc :: BoardState -> Int -> Int
countDisc bs player = sum [ length (filter (==player) row) | row <- board bs ]

{-
checkWinner, return winner of board (1, 2), or 0 if tie
-}

checkWinner :: BoardState -> Int
checkWinner bs
    | p1Count > p2Count = 1
    | p2Count > p1Count = 2
    | otherwise = 0
    where
        p1Count = countDisc bs 1
        p2Count = countDisc bs 2

-- MCTS related functions start here
-- Tutorial on MCTS, where the following code snippets are based off:
    https://www.youtube.com/watch?v=UXW2yZndl7U&t=1s
```

```haskell
{-
MCTS Node data structure
-}
data MCTSNode = MCTSNode {
    state      :: BoardState,
    children   :: [MCTSNode],
    n_i        :: Int,
    t          :: Double,
    m          :: Maybe Position -- The move that led to this state; Nothing for root node so we
        use Maybe
} deriving (Show, Generic, NFData)

{-
Calculate UCT value for a given MCTSNode
-}
uctValue :: MCTSNode -> Int -> Double
uctValue node parentN =
    if n_i node == 0 then 1/0 -- infinity
    else (t node / fromIntegral (n_i node)) + (sqrt 2) * sqrt (log (fromIntegral parentN) /
        fromIntegral (n_i node))

{-
Selection: select the best child to perform simulation based on UCT values
Also keeps track of path from root to selected node for backpropagation later
-}
selection :: MCTSNode -> [MCTSNode] -> (MCTSNode, [MCTSNode])
selection node path =
    if null (children node)
    then (node, path)
    else selection bestChild (path ++ [node])
    where
        bestChild = maximumBy (comparing (\child -> uctValue child (n_i node))) (children node)



{-
Expansion: expand the selected node
For each avalable action from current, add a new state to the children of the node / to the tree
-}
expansion :: MCTSNode -> [Position] -> MCTSNode
expansion node []   = node -- No possible moves to expand any further
expansion node (x:xs) = expansion (node { children = newNode : children node }) xs
    where
        bsNew = updateTurn x (state node)
        newNode = MCTSNode {
            state = bsNew,
            children = [],
            n_i = 0,
            t = 0.0,
            m = Just x
        }

{-
Simulation: simulate a random playout from the given node's state until terminal state is reached
-}
simulation :: MCTSNode -> StdGen -> Double
simulation node gen = simulateFromState (state node) gen

simulateFromState :: BoardState -> StdGen -> Double
```

```
simulateFromState bs g =
    let moves = getPossibleMoves bs
    in if null moves
        then evaluateBoardMTCS bs
        else
            let (randomIndex, g') = randomR (0, length moves - 1) g
                move = moves !! randomIndex
                newBs = updateTurn move bs
            in simulateFromState newBs g'

evaluateBoardMTCS :: BoardState -> Double
evaluateBoardMTCS bs = fromIntegral (p2Count - p1Count)
    where
        p1Count = countDisc bs 1
        p2Count = countDisc bs 2

{-
Backpropagation: propagate n_i and t values along the path from selected node back to root
Also update children field of the node along the way
[MCTSNode] is of the path from leaf to root
           It is of the form [a, a's parent, a's grandparent, ...]
           So we use the previous node to update the children field of the current node
MCTSNode: previously updated node (the node we will use to update children field of current node
    in the path)
Double: reward obtained from simulation
MCTSNode: return the last node updated (AKA root node updated)
-}
backpropagation :: [MCTSNode] -> MCTSNode -> Double -> MCTSNode
backpropagation [] updatedNode _ = updatedNode
backpropagation (parent:rest) updatedChild reward =
    let updatedParent = parent {
            children = map (\c -> if m c == m updatedChild then updatedChild else c) (children
                parent),
            n_i = n_i parent + 1,
            t = t parent + reward
        }
    in backpropagation rest updatedParent reward

{-
Run one iteration of MCTS:
1. selection
2. expansion
3. simulation
4. backpropagation
-}
runMCTSIteration :: MCTSNode -> StdGen -> MCTSNode
runMCTSIteration root gen =
    -- 1. selection, select the best leaf, starting from the root
    let (selectedNode, path) = selection root [] -- selection returns leaf node in current MCTS
        tree
        possibleMoves = getPossibleMoves (state selectedNode)
    in  if null possibleMoves
        then
            let reward = evaluateBoardMTCS (state selectedNode) -- Terminal node: can directly
                evaluate without simulation
                selectedNode' = selectedNode {
                    n_i = n_i selectedNode + 1,
                    t = t selectedNode + reward
                }
```

```
            updatedRoot = backpropagation (reverse path) selectedNode' reward -- Still need to
                backpropagate the reward up the tree
        in updatedRoot
    else if n_i selectedNode == 0 && m selectedNode /= Nothing -- is n_i for current 0? Also
        need to make sure it's not root
        then
            -- Directly simulate/rollout from selectedNode without expansion
            let reward = simulation selectedNode (snd $ split gen) -- 3. Simulation
                selectedNode' = selectedNode {
                    n_i = n_i selectedNode + 1,
                    t = t selectedNode + reward
                }
                updatedRoot = backpropagation (reverse path) selectedNode' reward
            in updatedRoot
        else
            -- Expand selectedNode, then simulate/rollout from the new child
            let expandedNode = expansion selectedNode possibleMoves
                (childToSimulate, _) = selection expandedNode []
                reward = simulation childToSimulate (snd $ split gen)
                -- expandedNode is the parent of the updated child (childToSimulate')
                childToSimulate' = childToSimulate {
                    n_i = n_i childToSimulate + 1,
                    t = t childToSimulate + reward
                }
                updatedRoot = backpropagation (expandedNode : reverse path) childToSimulate'
                    reward
            in updatedRoot

{-
Run MCTS for a given number of iterations starting from the given root node
MCTSNode: root of tree
Int: number of iterations to run
StdGen: random generator for simulations
return new MCTSNode root where the children have been added/updated according to MCTS iterations
-}
runMCTS :: MCTSNode -> Int -> [StdGen] -> MCTSNode
runMCTS root _ [] = root
runMCTS root 0 _ = root
runMCTS root n (g:gs) = runMCTS (runMCTSIteration root g) (n - 1) gs


{- Version 2: split 1000 iterations into chunks of 10 iterations each, and run each chunk in
    parallel
let the runtime dynamically allocate threads to handle each chunk -}
runMCTSParallel :: MCTSNode -> Int -> StdGen -> MCTSNode
runMCTSParallel root iterations gen = mergedRoot
    where
        gens = take iterations $ iterate (snd . split) gen
        gensGrouped = chunksOf 10 gens
        updatedRoots = parMap rdeepseq (\g -> runMCTS root 10 g) gensGrouped
        mergedRoot = mergeMCTSRuns (tail updatedRoots) (head updatedRoots)


mergeMCTSRuns :: [MCTSNode] -> MCTSNode -> MCTSNode
mergeMCTSRuns [] root = root
mergeMCTSRuns (x:xs) root = mergeMCTSRuns xs mergedRoot
    where
        mergedRoot = root {
            n_i = n_i root + n_i x,
```

```
            t = t root + t x,
            children = mergeChildren (children root) (children x)
        }

mergeChildren :: [MCTSNode] -> [MCTSNode] -> [MCTSNode]
mergeChildren (x:xs) ys = mergedChild : mergeChildren xs ys
    where
        move = m x
        y = head $ filter (\child -> m child == move) ys -- find corresponding child in ys based on
            move
        mergedChild = x {
            n_i = n_i x + n_i y,
            t = t x + t y,
            m = move, -- both children should have same move
            children = [] -- We don't need to go deeper since we only care about root's children
                for MCTS decision making
        }
mergeChildren _ _ = [] -- children root and children x should be the same, except each child's n_i
    and t
                    -- so both lists should have same length, and we don't need to worry about
                        mismatch in length
-- MCTS related functions end here


{-
noPrintGameLoop logic for alternating between players without printing boards
-}
noPrintGameLoop :: BoardState -> IO ()
noPrintGameLoop bs = do
    let possibleMoves = getPossibleMoves bs
    if null possibleMoves
        then do
            let oppPlayer = if curr_player bs == 1 then 2 else 1
                oppPossibleMoves = getPossibleMoves (BoardState { board = board bs, curr_player =
                    oppPlayer })
            if null oppPossibleMoves
                then do
                    let winner = checkWinner bs
                    if winner == 0
                    then putStrLn "Game over! It's a tie!"
                    else do
                        putStrLn $ "Game over! Winner is Player " ++ show winner
                        putStrLn $ "Player 1 discs: " ++ show (countDisc bs 1)
                        putStrLn $ "Player 2 discs: " ++ show (countDisc bs 2)
                else do
                    putStrLn $ "Player " ++ show (curr_player bs) ++ " has no moves. Skipping turn."
                    noPrintGameLoop (BoardState { board = board bs, curr_player = oppPlayer })
    else do
        move <- if curr_player bs == 1
                    then do
                        -- Pick a random move for player 1
                        gen <- newStdGen
                        let (randomIndex, _) = randomR (0, length possibleMoves - 1) gen
                        let move = possibleMoves !! randomIndex
                        return move
                    else do
                        let rootNode = MCTSNode {
                            state = bs,
                            children = [],
```

```
                        n_i = 0,
                        t = 0.0,
                        m = Nothing
                    }
                    gen <- newStdGen
                    -- Attempt 2
                    let mctsRoot = runMCTSParallel rootNode 1000 gen
                    let bestChild = maximumBy (comparing (\child -> uctValue child (n_i
                        mctsRoot))) (children mctsRoot)
                        move = case m bestChild of
                                  Just pos -> pos
                                  Nothing -> error "No move found. Something went wrong with
                                      MCTS implementation."
                    return move
        let newGameState = updateTurn move bs
        noPrintGameLoop newGameState

main :: IO ()
main = do
    noPrintGameLoop initializeBoard
```

## B.4 othello-mcts-par-v3.hs

```
{-# LANGUAGE DeriveAnyClass #-}
{-# LANGUAGE DeriveGeneric #-}
{-# OPTIONS_GHC -Wno-unrecognised-pragmas #-}
{-# HLINT ignore "Avoid lambda" #-}
import System.Random (StdGen, newStdGen, randomR, split)
import Data.List (maximumBy)
import Data.List.Split (chunksOf)
import Data.Ord (comparing)
import Control.Parallel.Strategies
import GHC.Generics (Generic)

{-
This file contains logic for chunked root parallelized with
additional parallelized tree mergingMCTS (version 3)

Commands to compile and run this program:
stack install random
stack install split

stack ghc --package random -- -Wall -O2 -threaded -rtsopts -o othello-mcts-par-v3
    othello-mcts-par-v3.hs
./othello-mcts-par-v3 +RTS -N2 -s -l
./threadscope othello-mcts-par-v3.eventlog
-}

-- Game types
type Position = (Int, Int)

type Board = [[Int]] -- 0: empty, 1: player 1's disk, 2: player 2's disk

data BoardState = BoardState {
    board      :: Board,
    curr_player :: Int
} deriving (Show, Generic, NFData)
```

```haskell
rows :: Int
rows = 8

cols :: Int
cols = 8

gameBoard :: Board
gameBoard = [[0 | _ <- [1..cols]] | _ <- [1..rows]] :: [[Int]]

updateBoardIndex :: Board -> Position -> Int -> Board
updateBoardIndex board (i,j) val =
    case splitAt i board of
        (prevRows, currRow : afterRows) ->
            case splitAt j currRow of
                (prevElems, _ : afterElems) ->
                    prevRows ++ [prevElems ++ [val] ++ afterElems] ++ afterRows
                (_, _) -> board
        (_, _) -> board

updateBoardIndexes :: Board -> [Position] -> [Int] -> Board
updateBoardIndexes board (x:xs) (y:ys) = updateBoardIndexes newBoard xs ys where
    newBoard = updateBoardIndex board x y
updateBoardIndexes board _ _ = board

{-
Initialize the gameboard to start configuration and set current player to 1
-}
initializeBoard :: BoardState
initializeBoard = BoardState {
    board = updateBoardIndexes gameBoard [(3,3),(4,4),(4,3),(3,4)] [1,1,2,2],
    curr_player = 1
}

{-
Return the number at index (i,j) of the board
-}
getBoardVal :: Board -> Position -> Int
getBoardVal board (i,j) = (board !! i) !! j

{-
Defined 8-way directions for checking adjacency and flanking rules
-}
directions :: [Position]
directions = [(-1,-1), (-1,0), (-1,1), (0,-1), (0,1), (1,-1), (1,0), (1,1)]

{-
Check if a given position is within bounds of the board
-}
inBounds :: Position -> Bool
inBounds (x,y) = x >= 0 && x < rows && y >= 0 && y < cols

{-
Given a board state, return the possible next moves for the player
-}
getPossibleMoves :: BoardState -> [Position]
getPossibleMoves bs = [ (i,j) | i <- [0..rows-1],
                            j <- [0..cols-1],
                            getBoardVal (board bs) (i,j) == 0, -- spot is not taken yet
```

```
                        not (null (adjacencyAndFlankingCheck bs (i,j))) -- adjacency and
                            flanking rules are both met
                        ]

{-
Check if the adjacency and flanking conditions are both met for a given position
Return a list of directions where both conditions are met
If the list is non-empty, then the position is a valid move that satisfy both adjancy and
    flankinng rules
-}
adjacencyAndFlankingCheck :: BoardState -> Position -> [Position]
adjacencyAndFlankingCheck bs (i,j) = [ d | d <- directions, isAdjacentToOpp d && flankOpp d]
    where
        b = board bs
        curPlayer = curr_player bs
        oppPlayer = if curPlayer == 1 then 2 else 1

        isAdjacentToOpp (di,dj) = inBounds (i+di,j+dj) && getBoardVal b (i+di,j+dj) == oppPlayer

        {-
        Because of short-circuiting nature of &&, flankOpp will only be called if isAdjacentToOpp
            is True
        for the given direction (di,dj). As such, inside flankOpp, we can simplify checking
            flanking condition
        to be scanning towards direction (di,dj) and returning True if we find a disc of current
            player before hitting
        an empty spot or going out of bounds. Hitting a disc of current player means flanking
            condition is met in the
        (di,dj) direction because opponent's disc(s) is sandwiched between the new position (i,j)
            current player is
        going to take and current player's existing disc on board.
        -}
        flankOpp (di,dj) = scan (i + di + di) (j + dj + dj)
            where
                scan x y =
                    if not (inBounds (x,y)) then False
                    else case getBoardVal b (x,y) of
                        val | val == curPlayer -> True
                        val | val == oppPlayer -> scan (x + di) (y + dj)
                        _ -> False

{-
Given the position of the to-be-placed disc of current player and BoardState, update the discs on
    board and change curr_player to opponent player
to return an up-to-date BoardState to keep the game going
-}
updateTurn :: Position -> BoardState -> BoardState
-- Place a disc at `pos` for `curPlayer` and flip any sandwiched opponent disks
updateTurn (i,j) bs = BoardState {
    board = updatedBoard,
    curr_player = oppPlayer
}
    where
        b = board bs
        curPlayer = curr_player bs
        oppPlayer = if curPlayer == 1 then 2 else 1
        -- Figure out which directions are valid for flipping
        flippableDirections = adjacencyAndFlankingCheck bs (i,j)
```

```
        -- Helper function to get positions of all opponent discs to be flipped in one direction
            (di,dj)
        flipInDirection (di,dj) = scan (i + di) (j + dj) []
            where
                scan x y acc =
                    if not (inBounds (x,y)) then []
                    else case getBoardVal b (x,y) of
                        val | val == oppPlayer -> scan (x + di) (y + dj) ((x,y):acc)
                        val | val == curPlayer -> acc
                        _ -> []
        allPositionsToFlip = concatMap flipInDirection flippableDirections
        -- Replace index (i,j) and allPositionsToFlip with curPlayer
        updatedBoard = updateBoardIndexes b ((i,j):allPositionsToFlip) (replicate (1 + length
            allPositionsToFlip) curPlayer)

{-
Count number of discs of given player on board
-}
countDisc :: BoardState -> Int -> Int
countDisc bs player = sum [ length (filter (==player) row) | row <- board bs ]

{-
checkWinner, return winner of board (1, 2), or 0 if tie
-}

checkWinner :: BoardState -> Int
checkWinner bs
    | p1Count > p2Count = 1
    | p2Count > p1Count = 2
    | otherwise = 0
    where
        p1Count = countDisc bs 1
        p2Count = countDisc bs 2

-- MCTS related functions start here
-- Tutorial on MCTS, where the following code snippets are based off:
    https://www.youtube.com/watch?v=UXW2yZndl7U&t=1s
{-
MCTS Node data structure
-}
data MCTSNode = MCTSNode {
    state      :: BoardState,
    children   :: [MCTSNode],
    n_i        :: Int,
    t          :: Double,
    m          :: Maybe Position -- The move that led to this state; Nothing for root node so we
        use Maybe
} deriving (Show, Generic, NFData)

{-
Calculate UCT value for a given MCTSNode
-}
uctValue :: MCTSNode -> Int -> Double
uctValue node parentN =
    if n_i node == 0 then 1/0 -- infinity
    else (t node / fromIntegral (n_i node)) + (sqrt 2) * sqrt (log (fromIntegral parentN) /
        fromIntegral (n_i node))

{-
```

```
Selection: select the best child to perform simulation based on UCT values
Also keeps track of path from root to selected node for backpropagation later
-}
selection :: MCTSNode -> [MCTSNode] -> (MCTSNode, [MCTSNode])
selection node path =
    if null (children node)
    then (node, path)
    else selection bestChild (path ++ [node])
    where
        bestChild = maximumBy (comparing (\child -> uctValue child (n_i node))) (children node)



{-
Expansion: expand the selected node
For each avalable action from current, add a new state to the children of the node / to the tree
-}
expansion :: MCTSNode -> [Position] -> MCTSNode
expansion node []   = node -- No possible moves to expand any further
expansion node (x:xs) = expansion (node { children = newNode : children node }) xs
    where
        bsNew = updateTurn x (state node)
        newNode = MCTSNode {
            state = bsNew,
            children = [],
            n_i = 0,
            t = 0.0,
            m = Just x
        }

{-
Simulation: simulate a random playout from the given node's state until terminal state is reached
-}
simulation :: MCTSNode -> StdGen -> Double
simulation node gen = simulateFromState (state node) gen

simulateFromState :: BoardState -> StdGen -> Double
simulateFromState bs g =
    let moves = getPossibleMoves bs
    in if null moves
        then evaluateBoardMTCS bs
        else
            let (randomIndex, g') = randomR (0, length moves - 1) g
                move = moves !! randomIndex
                newBs = updateTurn move bs
            in simulateFromState newBs g'

evaluateBoardMTCS :: BoardState -> Double
evaluateBoardMTCS bs = fromIntegral (p2Count - p1Count)
    where
        p1Count = countDisc bs 1
        p2Count = countDisc bs 2

{-
Backpropagation: propagate n_i and t values along the path from selected node back to root
Also update children field of the node along the way
[MCTSNode] is of the path from leaf to root
           It is of the form [a, a's parent, a's grandparent, ...]
           So we use the previous node to update the children field of the current node
```

```
MCTSNode: previously updated node (the node we will use to update children field of current node
    in the path)
Double: reward obtained from simulation
MCTSNode: return the last node updated (AKA root node updated)
-}
backpropagation :: [MCTSNode] -> MCTSNode -> Double -> MCTSNode
backpropagation [] updatedNode _ = updatedNode
backpropagation (parent:rest) updatedChild reward =
    let updatedParent = parent {
            children = map (\c -> if m c == m updatedChild then updatedChild else c) (children
                parent),
            n_i = n_i parent + 1,
            t = t parent + reward
        }
    in backpropagation rest updatedParent reward

{-
Run one iteration of MCTS:
1. selection
2. expansion
3. simulation
4. backpropagation
-}
runMCTSIteration :: MCTSNode -> StdGen -> MCTSNode
runMCTSIteration root gen =
    -- 1. selection, select the best leaf, starting from the root
    let (selectedNode, path) = selection root [] -- selection returns leaf node in current MCTS
        tree
        possibleMoves = getPossibleMoves (state selectedNode)
    in  if null possibleMoves
        then
            let reward = evaluateBoardMTCS (state selectedNode) -- Terminal node: can directly
                evaluate without simulation
                selectedNode' = selectedNode {
                    n_i = n_i selectedNode + 1,
                    t = t selectedNode + reward
                }
                updatedRoot = backpropagation (reverse path) selectedNode' reward -- Still need to
                    backpropagate the reward up the tree
            in updatedRoot
        else if n_i selectedNode == 0 && m selectedNode /= Nothing -- is n_i for current 0? Also
            need to make sure it's not root
            then
                -- Directly simulate/rollout from selectedNode without expansion
                let reward = simulation selectedNode (snd $ split gen) -- 3. Simulation
                    selectedNode' = selectedNode {
                        n_i = n_i selectedNode + 1,
                        t = t selectedNode + reward
                    }
                    updatedRoot = backpropagation (reverse path) selectedNode' reward
                in updatedRoot
            else
                -- Expand selectedNode, then simulate/rollout from the new child
                let expandedNode = expansion selectedNode possibleMoves
                    (childToSimulate, _) = selection expandedNode []
                    reward = simulation childToSimulate (snd $ split gen)
                    -- expandedNode is the parent of the updated child (childToSimulate')
                    childToSimulate' = childToSimulate {
                        n_i = n_i childToSimulate + 1,
```

```
                    t = t childToSimulate + reward
                }
                updatedRoot = backpropagation (expandedNode : reverse path) childToSimulate'
                    reward
            in updatedRoot

{-
Run MCTS for a given number of iterations starting from the given root node
MCTSNode: root of tree
Int: number of iterations to run
StdGen: random generator for simulations
return new MCTSNode root where the children have been added/updated according to MCTS iterations
-}
runMCTS :: MCTSNode -> Int -> [StdGen] -> MCTSNode
runMCTS root _ [] = root
runMCTS root 0 _ = root
runMCTS root n (g:gs) = runMCTS (runMCTSIteration root g) (n - 1) gs


{-
Version 3: split 1000 iterations into chunks of 10 iterations each, and run each chunk in parallel
let the runtime dynamically allocate threads to handle each chunk. Call mergeMTCRunsParallel
-}
runMCTSParallel :: MCTSNode -> Int -> StdGen -> MCTSNode
runMCTSParallel root iterations gen = mergedRoot
    where
        gens = take iterations $ iterate (snd . split) gen
        gensGrouped = chunksOf 10 gens
        updatedRoots = parMap rdeepseq (\g -> runMCTS root 10 g) gensGrouped
        mergedRoot = mergeMCTSRunsParallel updatedRoots


mergeMCTSRunsParallel :: [MCTSNode] -> MCTSNode
mergeMCTSRunsParallel [] = error "Something is wrong with mergeMCTSRunsParallel" -- Should not
    really happen
mergeMCTSRunsParallel [x] = x
mergeMCTSRunsParallel xs = mergeMCTSRunsParallel mergedChunks
    where
        chunks = chunksOf 10 xs
        mergedChunks = parMap rdeepseq (\c -> mergeMCTSRuns (tail c) (head c)) chunks

mergeMCTSRuns :: [MCTSNode] -> MCTSNode -> MCTSNode
mergeMCTSRuns [] root = root
mergeMCTSRuns (x:xs) root = mergeMCTSRuns xs mergedRoot
    where
        mergedRoot = root {
            n_i = n_i root + n_i x,
            t = t root + t x,
            children = mergeChildren (children root) (children x)
        }

mergeChildren :: [MCTSNode] -> [MCTSNode] -> [MCTSNode]
mergeChildren (x:xs) ys = mergedChild : mergeChildren xs ys
    where
        move = m x
        y = head $ filter (\child -> m child == move) ys -- find corresponding child in ys based on
            move
        mergedChild = x {
            n_i = n_i x + n_i y,
```

```
            t = t_x + t_y,
            m = move, -- both children should have same move
            children = [] -- We don't need to go deeper since we only care about root's children
                for MCTS decision making
        }
mergeChildren _ _ = [] -- children root and children x should be the same, except each child's n_i
    and t
                    -- so both lists should have same length, and we don't need to worry about
                        mismatch in length
-- MCTS related functions end here


{-
noPrintGameLoop logic for alternating between players without printing boards
-}
noPrintGameLoop :: BoardState -> IO ()
noPrintGameLoop bs = do
    let possibleMoves = getPossibleMoves bs
    if null possibleMoves
        then do
            let oppPlayer = if curr_player bs == 1 then 2 else 1
                oppPossibleMoves = getPossibleMoves (BoardState { board = board bs, curr_player =
                    oppPlayer })
            if null oppPossibleMoves
                then do
                    let winner = checkWinner bs
                    if winner == 0
                    then putStrLn "Game over! It's a tie!"
                    else do
                        putStrLn $ "Game over! Winner is Player " ++ show winner
                        putStrLn $ "Player 1 discs: " ++ show (countDisc bs 1)
                        putStrLn $ "Player 2 discs: " ++ show (countDisc bs 2)
                else do
                    putStrLn $ "Player " ++ show (curr_player bs) ++ " has no moves. Skipping turn."
                    noPrintGameLoop (BoardState { board = board bs, curr_player = oppPlayer })
    else do
        move <- if curr_player bs == 1
                    then do
                        -- Pick a random move for player 1
                        gen <- newStdGen
                        let (randomIndex, _) = randomR (0, length possibleMoves - 1) gen
                        let move = possibleMoves !! randomIndex
                        return move
                    else do
                        let rootNode = MCTSNode {
                            state = bs,
                            children = [],
                            n_i = 0,
                            t = 0.0,
                            m = Nothing
                        }
                        gen <- newStdGen
                        let mctsRoot = runMCTSParallel rootNode 1000 gen
                        let bestChild = maximumBy (comparing (\child -> uctValue child (n_i
                            mctsRoot))) (children mctsRoot)
                            move = case m bestChild of
                                        Just pos -> pos
                                        Nothing -> error "No move found. Something went wrong with
                                            MCTS implementation."
```

```
                    return move
        let newGameState = updateTurn move bs
        noPrintGameLoop newGameState

main :: IO ()
main = do
    noPrintGameLoop initializeBoard
```