Othello

Noam Hirschorn (nyh2111) & Dan Ivanovich (dmi2115)

Background

Othello, a board game derived from Reversi, is played on an 8x8 grid with a fixed starting layout. Throughout the game, players take turns placing disks of their side's color onto empty spaces. After a player places a disk, any disks of the opponent's color that lie between the one that was just placed and another one of the same color are flipped. At the end of the game (when the board is filled), whichever player has the majority of disks showing their color wins the game. The game can end in a tie if both players have the same number of disks when there are no remaining moves to be made.

Minimax (AKA minmax) is a back-tracking decision-making algorithm that is often used to determine the optimal move in turn-based games (eg. chess, checkers, tic-tac-toe, or othello). It assumes that the opponent will play optimally and recursively evaluates all future states of the board, then selects the move that eventually leads to the most favorable outcome. By developing an algorithm to assign a score to each board state that reflects which player holds the most advantageous position, one can use minimax to create an intelligent opponent that optimally plays towards a winning result. While minimax is a great framework for decision-making, it is limited by the exponential growth of possible game states as search depth increases, making deeper searches computationally expensive, particularly for games like Othello with high branching factors. To address this, practical implementations often limit the search depth and rely on heuristic evaluation functions to approximate the desirability of non-terminal board states. In the case of Othello, an example simple heuristic is the number of stones of the chosen color in a given state to estimate desirability.

Development

The first task we had was to adapt the codebase. Much of it involved setting up a GUI to allow for human vs AI play. We extracted the parts needed to actually make a move, and just had it load a given game board, perform a search, and return a move. This allowed for much more accurate measuring of the minmax algorithm for a single search tree. Additionally, we added a simple alpha beta functionality to the search.

Our first attempt at parallelization had issues. Specifically, the threads would be sequentially dependent on each other, which was due to incorrect handling of alpha beta parameters, so 2 threads would not run simultaneously as they would wait for alpha beta values to be updated. Once we identified this issue with threadscope, we created two models of parallelization. One would simply try to parallelize the top few layers of the search tree until a given depth, and have each thread run in isolation below it (known as the top-down method). The other involved intentionally causing threads to only parallelize at a lower depth where each node there would have all its children expanded in parallel. The threads would then run, and when all complete, update the alpha beta value, and move on to the next node in that level (known as the side-side method). This algorithm was meant to allow for additional use of sequential moves in the top few levels. We found that the first method merged into the second, since having multiple levels of parallelization would lead to excess fizzling as the parallelized upper layers would attempt to redo the work of parallelized lower layers (so this ‘top-down’ case simply became a specialized ‘side-side’ case with the ‘parallelDepth’, the level nodes would be parallelized from, set to the top layer). The spirit of the top-down theory would involve using BFS to get to a desired parallelDepth, and then have each node be sparked and run in parallel. However, BFS goes against the way alpha beta works, so this line of thought did not end up bearing fruit. An additional tactic used was ‘rdeepseq’ and ‘!’ to force evaluations before values were returned. This prevented thunks from building up in the parallelization step before needing to actually be evaluated when sequentially using the values to find the answer.

Another method attempted was to try to explicitly call numCapabilities to assign work based on the number of threads. However, this did not appear to help performance. Finally, chunks were used to try to make sure each thread had enough work to do, particularly when parallelDepth was set to lower levels of the search tree. Using chunksOf were both tried, but again had similar performance. At this point, there was some improvement from 1 thread to 2, but not much afterwards. In hindsight, while chunking may be helpful for unbalanced search trees, since Othello tends to be balanced, it merely adds to overhead.

An attempt was made to not force threads to wait for each other; with parMap, each thread must finish the sparks of the node they were expanding before the code can move on to the next node at parallelDepth to expand. Instead, the goal was to have threads allowed to immediately continue to the next node which would need expansion with the alpha beta values currently calculated. However, this would require thread communication as threads would need to update alpha beta values independently, and generally need more communication to allow them to work so independently. Moreover, it would be difficult to stop a thread once it started working on the next node if that ended up not needing to be expanded. Ultimately, this avenue was dropped as the code required was getting convoluted and appeared likely to degrade performance with all of the synchronization. Indeed, an analysis of threadscope showed that very few cycles were missed from a faster thread waiting for a slower one since the search tree is fairly balanced.

A small breakthrough was made with the use of parBuffer, as well as the basic map function. This allowed for additional balancing of tasks in case the search tree can be unbalanced or the number of threads is not equal to the number of sparks to be created at that node.At first, a fixed number was used to determine the buffer size, but it then was changed to be a function of the number of threads. These methods generally saw an increase in cost from 1 thread to 2 due to the overhead of parallelization and setting up the buffer (1 thread didn’t involve actually working with sparks/they all fizzled anyways, but with 2, all of the overhead existed of the buffer and parallelization without much payoff). However, this code was then able to have better performance when the number of threads increased later.

A final tactic attempted was to parallelize moves by splitting them into subchunks using parListChunk. This method did not end up panning out (in general, manual chunking did not appear to have much impact), but this was not immediately clear. A user error was made when testing this functionality, as a “basic” version of parallelization was still present in the code (which served as a jumping off point before trying different techniques). The function call in the minimax function for the code about parallelizing evaluation was accidentally never updated, so tests purportedly using parListChunk were actually using a fairly basic method merely with parMap. Results seemed to be promising, so there was some tinkering with the exact nature of parListChunk which appeared to have some minor impacts (although any variance was actually due to noise, of which there is a considerable amount). This error was only discovered when preparing for the report, so although the function call itself was corrected (which revealed that the parListChunk function itself was not helpful), there was no time for alternative tries. Nevertheless, even the basic parMap code did produce statistically significant results, with decreases until N=4-6. In general, the two clear best performing approaches were to use parMap or parBuffer with a regular map. In the latter case, the specific buffer size used was set to 2 times the number of threads in order to ensure the buffer would be big enough, without overly wasting space by making it too large.

As a final note- turning off alpha beta pruning did not appear to greatly increase the relative performance as swept from 1 to n threads. This suggests that it might be some other factor (Amdahl’s law, sequential parts of the source, etc) which is preventing full use of the parallelization rather than the costs of ignoring alpha beta at the level when parallelizing. The results from both the parMap and parBuffer versions can be reversed to imply only about 25% of the code was parallelizable (slightly higher in the latter case), a trend consistent across thread numbers and depths.

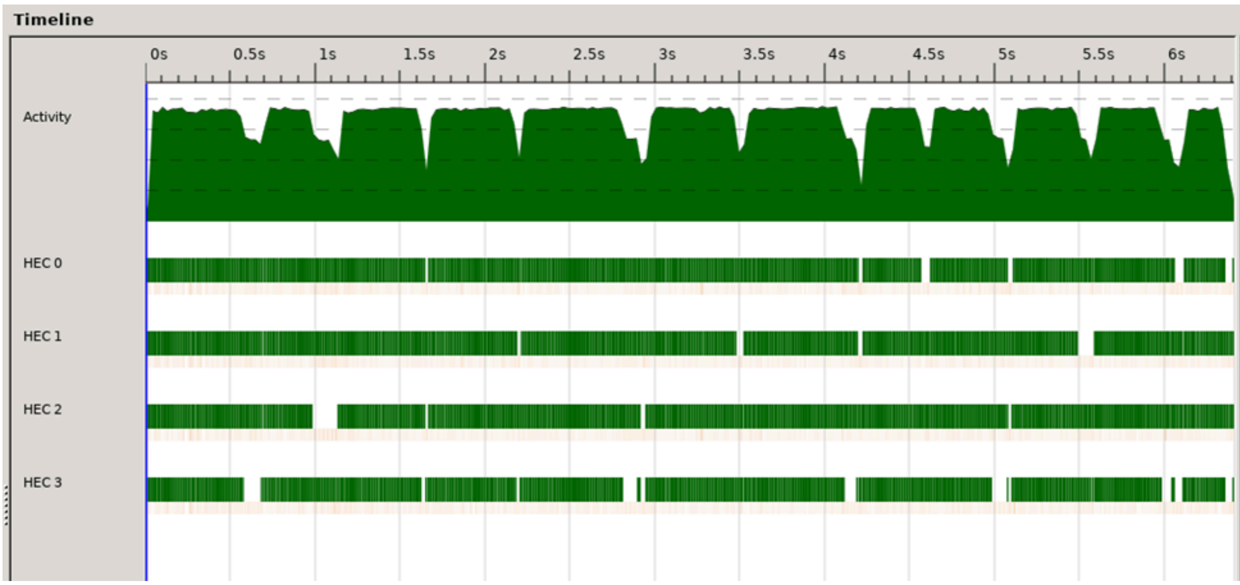
$$S = \frac{1}{(1-P) + \frac{P}{N}}$$

$$\frac{P}{N} - P = \frac{1}{S} - 1$$

$$P = \frac{\frac{1}{S} - 1}{\frac{1}{N} - 1} = \frac{\frac{N}{S} - N}{1 - N} = \frac{N - \frac{N}{S}}{N - 1}$$

E.g., for parMap, when N=4, S = .798(runtime N=1)/.637(runtimeN=4) = 1.252, P = 0.267

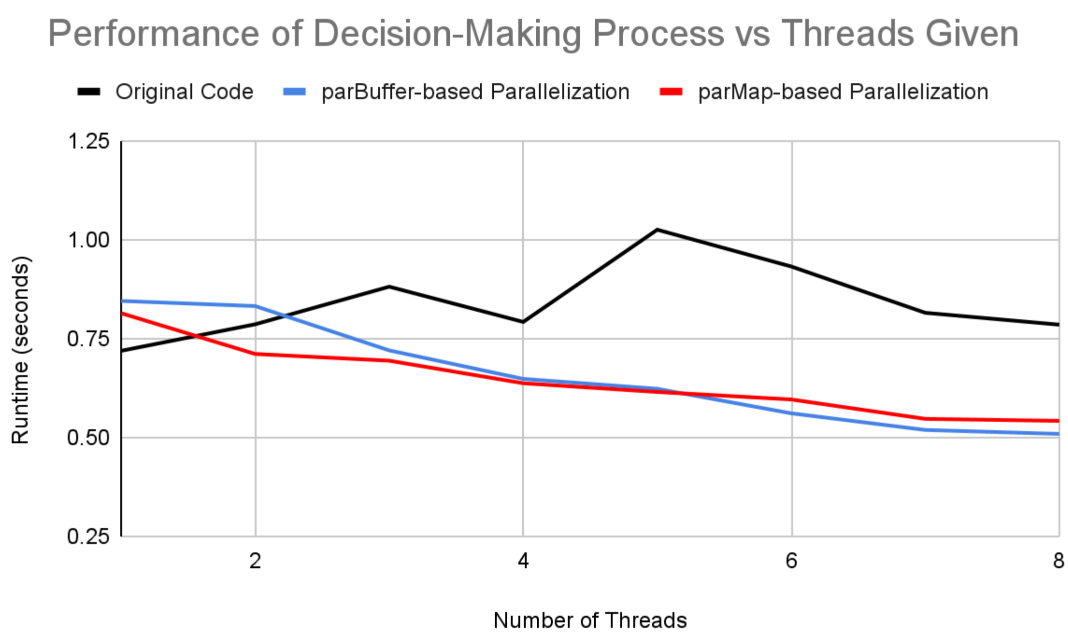
Nevertheless, the threadscope graph (below) would appear to indicate that all threads do tend to be active in parallel almost all of the time, with about 90% of sparks being converted. (Also worth noting is the fact that while it does appear there is some dead time as threads wait for each to finish, it does not appear to be a large amount/a driving cause). There is some amount of sequential setup which may be to blame (for example, loading the game file and printing results), but if this were the case, as the minimax tree was expanded the sequential effect would dissipate. Instead, there is likely some feature of the premade code which is causing the effects of parallelization to be muted (perhaps in the evaluation at the bottom layer of the graph, although that should just be done using the board in active memory).



Threadscope graph with depth = 6, parallelDepth =5, threads = 4

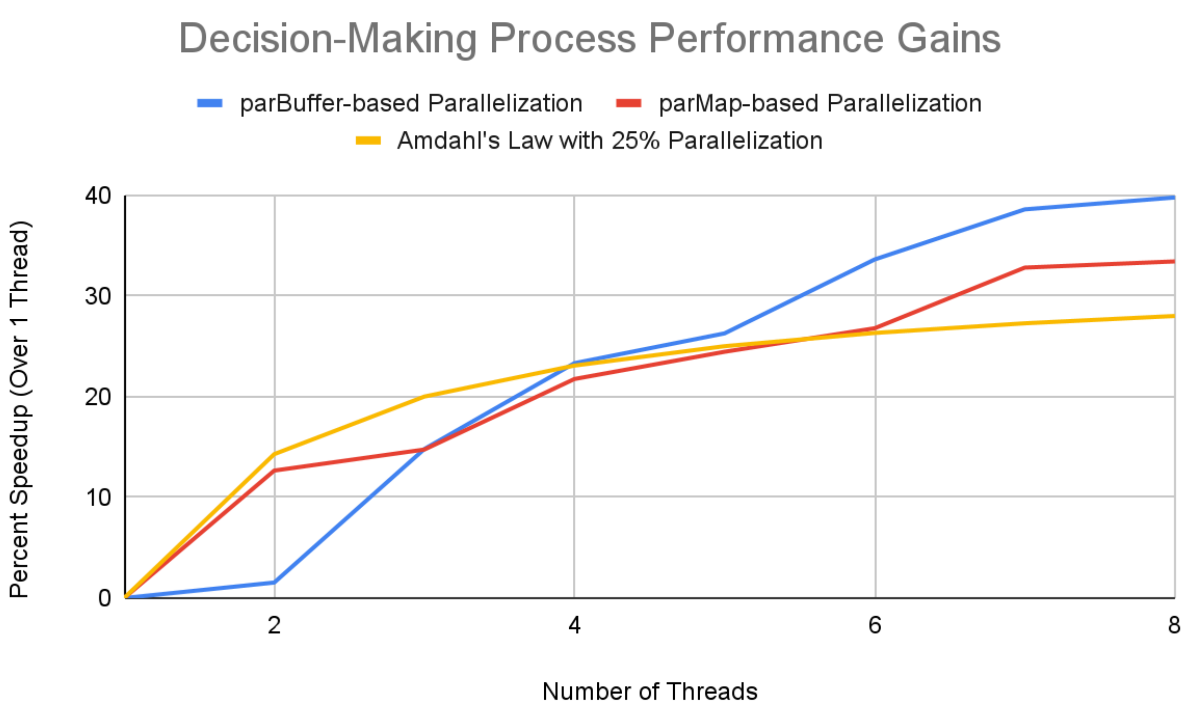
Performance

Throughout our many different approaches, we examined the performance while iterating over one of parallel depth, depth, and the number of threads given, and holding all else constant. Ultimately, we only discovered meaningful correlations between performance and the number of threads used by the program, with results remaining constant across other sampled parameters. We found that our parBuffer-based Parallelization (side_side_7.hs) and our parMap-based Parallelization (side_side_9.hs) proved to be the most consistent high performers. A factor we sampled over included varying the different starting states of the board, with some having less moves and others having more possible moves (and therefore a larger search tree), and found that the parBuffer-based Parallelization seemed to beat parMap when it came to performance on smaller search trees, and parMap performed better on larger search trees. Depending on the starting state of the board though, both methods would sometimes see that running with 2 threads would result in a slightly longer runtime than 1 thread, before continuing to improve with each added thread as normal—this could be due to the overhead of initializing parallel threads outweighing the computational savings on smaller search trees, where the workload is insufficient to fully utilize multiple threads effectively. In general, when given at least 3 threads, both of these approaches are able to non-negligibly output the best performance of the original code:



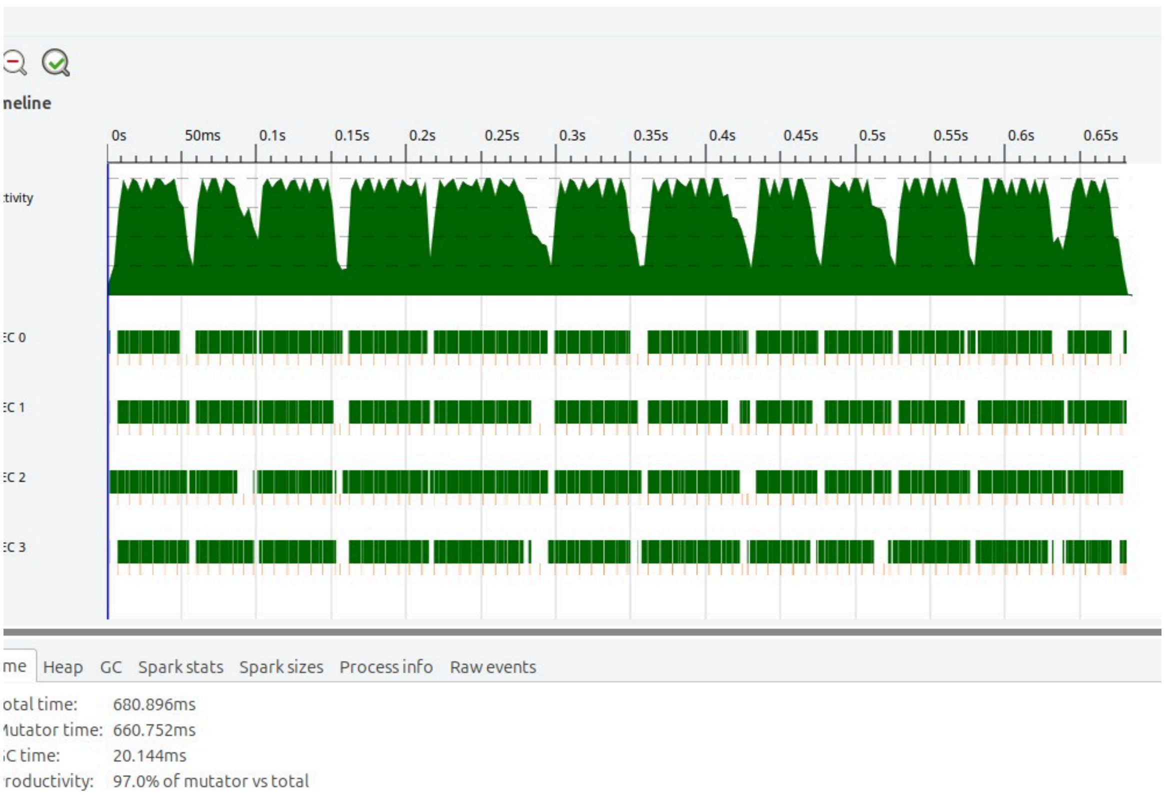
All data collected while running at depth = 5, parallelDepth =4, starting board = custom_game_2

While these performance gains over the original code are significant, the intention behind both the parMap and parBuffer approaches is clear when you compare their performance at each thread count against its single-threaded execution time: When given at least 3 threads, both of these approaches are able to non-negligibly output the best performance of the original code.

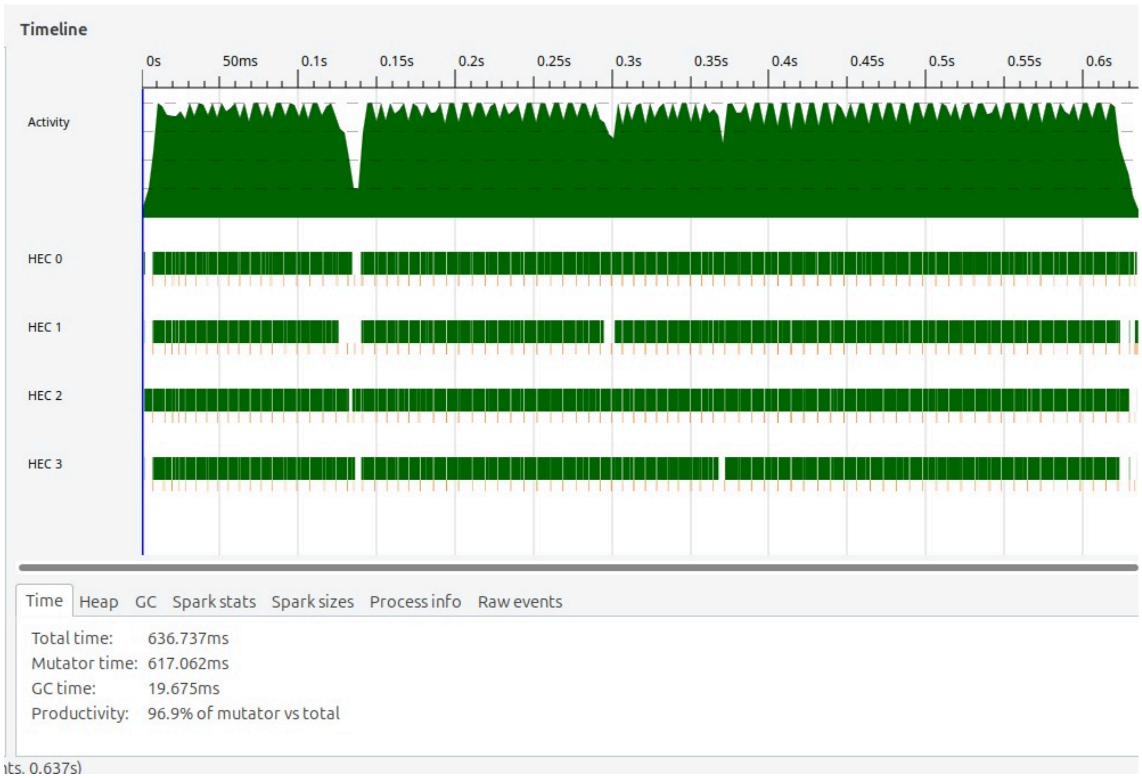


All data collected while running at depth = 5, parallelDepth =4, starting board = custom_game_2

There is a clear relationship between the number of threads used and the performance gains seen, with significant performance improvements seen at each increment. One could say that the parBuffer-based approach “starts slower,” then more rapidly sees improvements in performance as more threads are used. The parMap-based parallelization, on the other hand, shows a closer adherence to a linear relationship between thread count and improvement. As is demonstrated by the graphs below, the parBuffer allows for a better queue for sparks/work to build up in, so there is less downtime, although both do a fairly good job at keeping threads active. By creating a solid relationship between thread count and performance gains, both of these approaches achieved our goal of demonstrably parallelizing the original algorithm, while also meaningfully outperforming the original code. Each of these methods also resulted in a fairly well-balanced load, with the work being distributed well:

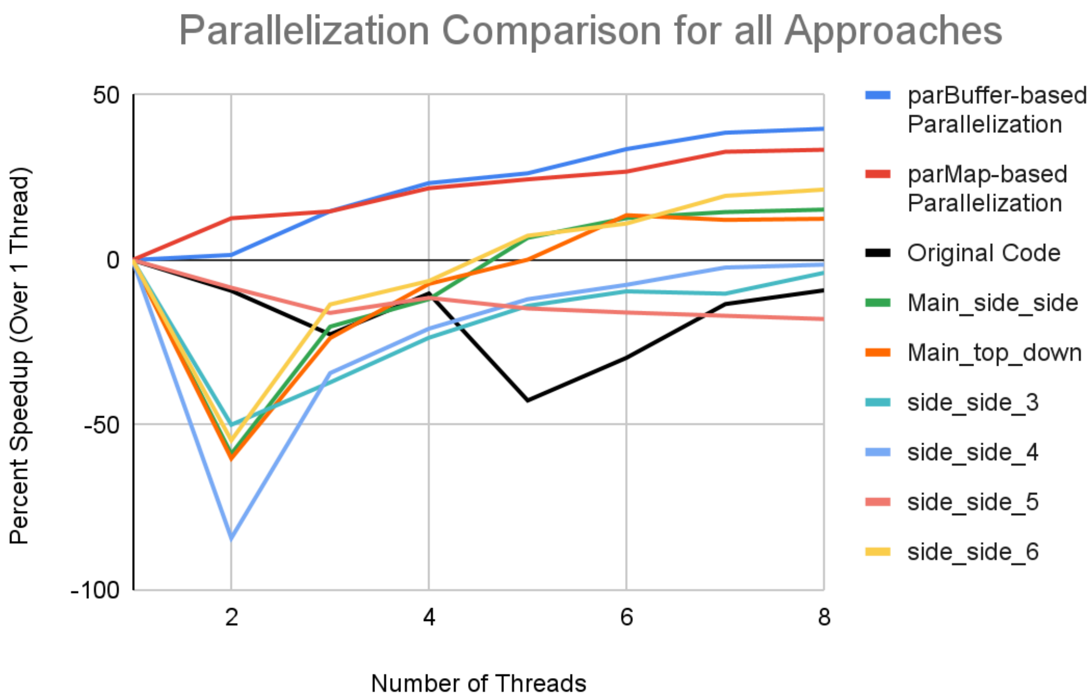


Threadscope graph for the parMap-based parallelization with 4 threads



Threadscope graph for the parBuffer-based parallelization with 4 threads

It's also interesting to see the comparative successes and failures of our many parallelization attempts, and our progression towards our best results,on the same graph. There is a clear common drop when parallelization must be set up at n=2, but then a recovery to various extents. The key difference between with our better performing versions is this lack of initial drop. The visualization does provide additional context for the aforementioned drop of parBuffer (albeit to a far lesser extent than other versions), making parMap appear unique for not suffering this drop.



Conclusion

Our attempts to parallelize the minimax algorithm for the Othello player were eventually able to demonstrate clear performance improvements in both parMap-based and parBuffer-based approaches. While each of these methods has its strengths—parMap showing a more consistent, linear scaling with thread count and parBuffer performing better with higher thread counts—their combined results underscore the potential for meaningful parallelization. Despite there being inherently sequential components of the original code, both approaches achieved measurable gains in efficiency, successfully balancing workload across threads and outperforming the original implementation.

References

- Initial codebase: https://arttuys.fi/coding/2022/05/othello-haskell/

Folder othello

17 printable files

(file list disabled)

othello/README.md

Othello

COMS 4995 Parallel Functional Programming Final Project

Noam Hirschorn (nyh2111) & Dan Ivanovich (dmi2115)

Our experiments in parallelizing the performance of an Othello minmax agent. Adapted from this codebase.

Building

This project uses `cabal`. To compile the code, simply run `cabal build` from the root directory (. in the file listing)

By default, this will build side_side9.hs, the parMap-based Parallelization. To build another file, update the following lin in hello-othello.cabal:

```
executable othello
    main-is:         side_side_9.hs  <-- Change this file name
```

and then run `cabal build` again.

Running

After running `cabal build`, run using `cabal run othello -- <how deep search tree should go> <what depth to start parallization at> <gameboard file> +RTS -N<number of threads> -s`.

Example:

```
$ cabal build
Resolving dependencies...
<Output Truncated For Brevity>

$ cabal run othello -- 5 4 custom_game_2.txt +RTS -N8 -s
Using minimax depth: 5
Parallelizing at depth: 4
Next move: (5,2)
<Output Truncated For Brevity>
```

Some suggested test case parameters:

- `cabal run othello -- 5 4 custom_game_2.txt +RTS -N3 -s`
- `cabal run othello -- 5 4 custom_game_2.txt +RTS -N6 -s`

- `cabal run othello -- 6 5 custom_game_2.txt +RTS -N3 -s`
- `cabal run othello -- 6 5 custom_game_2.txt +RTS -N6 -s`

File Listing

```
.
├── app
│   ├── Main_side_side.hs
│   ├── Main_top_down.hs
│   ├── side_side_3.hs
│   ├── side_side_4.hs
│   ├── side_side_5.hs
│   ├── side_side_6.hs
│   ├── side_side_7.hs    # One of our two best performers - parBuffer-based Parallelization
│   ├── side_side_8.hs
│   └── side_side_9.hs    # One of our two best performers - parMap-based Parallelization
├── benchmark_performance.py  # Performance benchmarking script, can run with --help for info
├── cabal.project
├── custom_game_1.txt     # A starting board with only a few possible moves
├── custom_game_2.txt     # A starting board with far more possible moves
├── haskell-othello.cabal # Adjust to pick which of the files in /app to build & run
├── README.md             # This file
└── src
    └── Othello
        ├── backup_Gamelog.hs
        └── GameLogic.hs
```

othello/app/Main_side_side.hs

```haskell
{-# LANGUAGE BangPatterns #-}

import System.Environment (getArgs)
import System.IO (readFile)
import Data.Array (array, listArray, assocs)
import Data.Foldable (maximumBy)
import Data.Ord (comparing)
import Control.Parallel.Strategies (parMap, rpar, rseq, using, parList, rdeepseq)
import Othello.GameLogic (
    GameSetup(..), Player(..), initialGameState, Board(..), DiscState(..),
    movesForPlayer, applyMove, Coordinate, opposingPlayer
    )

-- Parse a custom board state from a string
parseCustomBoard :: String -> Int -> Board
parseCustomBoard input dim = Board {
    grid = listArray ((0, 0), (dim-1, dim-1)) discStates,
    boardDim = dim
  }
  where
    rows = lines input
    discStates = concatMap parseRow rows
    parseRow row = map parseDiscState (words row)
    parseDiscState "E" = Empty
    parseDiscState "R" = Placed Red
```

```haskell
    parseDiscState "B" = Placed Blue
    parseDiscState _   = error "Invalid disc state in custom board. Use 'E', 'R', or 'B'."

-- Define a simple entry point for the program
main :: IO ()
main = do
    -- Get command-line arguments
    args <- getArgs

    -- Parse depth arguments
    let (depth, parallelDepth, inputFile) = case args of
            (d:p:file:_) -> (read d, read p, Just file)
            (d:p:_)      -> (read d, read p, Nothing)
            _            -> error "Usage: <depth> <parallelDepth> [gameboard file]"

    -- Load the custom board or use the default initial game state
    gameSetup <- case inputFile of
        Just file -> do
            content <- readFile file
            let customBoard = parseCustomBoard content 8 -- Assume an 8x8 board for simplicity
            return GameSetup {
                board = customBoard,
                aiPlays = [Red, Blue],
                searchDepth = depth
            }
        Nothing -> return (initialGameState 8 [Red, Blue] depth)

    putStrLn $ "Using minimax depth: " ++ show depth
    putStrLn $ "Parallelizing at depth: " ++ show parallelDepth

    let currentPlayer = Red

    let moves = getAIMove gameSetup currentPlayer depth parallelDepth
    if null moves
        then putStrLn "No valid moves available."
        else do
            let selectedMove = head moves
            let updatedBoard = applyMove (board gameSetup) currentPlayer [selectedMove]
            putStrLn $ "Next move: " ++ show selectedMove
            putStrLn "Updated board state:"
            print updatedBoard

-- Define a custom AI move function using alpha-beta pruning
getAIMove :: GameSetup -> Player -> Int -> Int -> [Coordinate]
getAIMove setup player depth parallelDepth
    | null possibleMoves = [] -- No moves available
    | otherwise = fst $ maximumBy (comparing snd) evaluatedMoves
  where
```

```
    possibleMoves = movesForPlayer (board setup) player

    evaluatedMoves =
        if depth >= parallelDepth
        then parallelEvaluate possibleMoves
        else sequentialEvaluate possibleMoves

    sequentialEvaluate moves =
        [ (move, minimax (applyMove (board setup) player move) (opposingPlayer player) (depth - 1) minBound maxBound)
        | move <- moves
        ]

    parallelEvaluate moves =
        let groupedMoves = groupByThread possibleMoves (length moves)
            evalGroup group =
                [ (move, minimax (applyMove (board setup) player move) (opposingPlayer player) (depth - 1) minBound maxBound)
                | (move, _) <- group
                ]
        in concat $ parMap rdeepseq evalGroup groupedMoves

    groupByThread moves numThreads =
        [ [ (move, idx) | (move, idx) <- zip moves [0..], idx `mod` numThreads == threadIdx ]
        | threadIdx <- [0..numThreads - 1]
        ]


-- Minimax algorithm with alpha-beta pruning
minimax :: Board -> Player -> Int -> Int -> Int -> Int
minimax board player depth alpha beta
    | depth == 0 || null possibleMoves = evaluateBoard board player --base case
    | player == maximizingPlayer = maximize alpha beta possibleMoves
    | otherwise = minimize alpha beta possibleMoves
  where
    possibleMoves = movesForPlayer board player
    maximizingPlayer = Red

    maximize :: Int -> Int -> [[Coordinate]] -> Int
    maximize a b [] = a
    maximize a b (move:moves)
        | a' >= b  = a'
        | otherwise = maximize a' b moves
      where
        a' = max a (minimax (applyMove board player move) (opposingPlayer player) (depth - 1) a b)

    minimize :: Int -> Int -> [[Coordinate]] -> Int
    minimize a b [] = b
    minimize a b (move:moves)
```

```haskell
        | a >= b' = b'
        | otherwise = minimize a b' moves
      where
        b' = min b (minimax (applyMove board player move) (opposingPlayer player) (depth - 1) a b)

-- Board evaluation
evaluateBoard :: Board -> Player -> Int
evaluateBoard board player = scoreForPlayer - scoreForOpponent
  where
    scoreForPlayer = length [pos | (pos, state) <- assocs (grid board), state == Placed player]
    scoreForOpponent = length [pos | (pos, state) <- assocs (grid board), state == Placed (opposingPlayer player)]



```

othello/app/Main_top_down.hs

```haskell
{-# LANGUAGE BangPatterns #-}

import System.Environment (getArgs)
import System.IO (readFile)
import Data.Array (array, listArray, assocs)
import Data.Foldable (maximumBy)
import Data.Ord (comparing)
import Control.Parallel.Strategies (parMap, rpar, rseq, using, parList, rdeepseq)
import Othello.GameLogic (
    GameSetup(..), Player(..), initialGameState, Board(..), DiscState(..),
    movesForPlayer, applyMove, Coordinate, opposingPlayer
    )

-- Parse a custom board state from a string
parseCustomBoard :: String -> Int -> Board
parseCustomBoard input dim = Board {
    grid = listArray ((0, 0), (dim-1, dim-1)) discStates,
    boardDim = dim
  }
  where
    rows = lines input
    discStates = concatMap parseRow rows
    parseRow row = map parseDiscState (words row)
    parseDiscState "E" = Empty
    parseDiscState "R" = Placed Red
    parseDiscState "B" = Placed Blue
    parseDiscState _   = error "Invalid disc state in custom board. Use 'E', 'R', or 'B'."

-- Define a simple entry point for the program
main :: IO ()
```

```haskell
main = do
    -- Get command-line arguments
    args <- getArgs

    -- Parse depth arguments
    let (depth, parallelDepth, inputFile) = case args of
            (d:p:file:_) -> (read d, read p, Just file)
            (d:p:_)      -> (read d, read p, Nothing)
            _            -> error "Usage: <depth> <parallelDepth> [gameboard file]"

    -- Load the custom board or use the default initial game state
    gameSetup <- case inputFile of
        Just file -> do
            content <- readFile file
            let customBoard = parseCustomBoard content 8 -- Assume an 8x8 board for simplicity
            return GameSetup {
                board = customBoard,
                aiPlays = [Red, Blue],
                searchDepth = depth
            }
        Nothing -> return (initialGameState 8 [Red, Blue] depth)

    putStrLn $ "Using minimax depth: " ++ show depth
    putStrLn $ "Parallelizing at depth: " ++ show parallelDepth

    let currentPlayer = Red
    let moves = getAIMove gameSetup currentPlayer depth parallelDepth
    if null moves
        then putStrLn "No valid moves available."
        else do
            let selectedMove = head moves
            let updatedBoard = applyMove (board gameSetup) currentPlayer [selectedMove]
            putStrLn $ "Next move: " ++ show selectedMove
            putStrLn "Updated board state:"
            print updatedBoard

-- Define a custom AI move function using alpha-beta pruning
getAIMove :: GameSetup -> Player -> Int -> Int -> [Coordinate]
getAIMove setup player depth parallelDepth
    | null possibleMoves = [] -- No moves available
    | otherwise = fst $ maximumBy (comparing snd) evaluatedMoves
  where
    possibleMoves = movesForPlayer (board setup) player

    evaluatedMoves =
        if depth >= parallelDepth
        then parallelEvaluate possibleMoves
        else sequentialEvaluate possibleMoves

```

```haskell
    sequentialEvaluate moves =
        [(move, minimax (applyMove (board setup) player move) (opposingPlayer player) (depth - 1) minBound maxBound)
        | move <- moves]

    parallelEvaluate moves =
        let eval move =
                (move, minimax (applyMove (board setup) player move) (opposingPlayer player) (depth - 1) minBound maxBound)
        in parMap rdeepseq eval moves

-- Minimax algorithm with alpha-beta pruning
minimax :: Board -> Player -> Int -> Int -> Int -> Int
minimax board player depth alpha beta
    | depth == 0 || null possibleMoves = evaluateBoard board player --Base case
    | player == maximizingPlayer = maximize alpha beta possibleMoves
    | otherwise = minimize alpha beta possibleMoves
  where
    possibleMoves = movesForPlayer board player
    maximizingPlayer = Red

    maximize :: Int -> Int -> [[Coordinate]] -> Int
    maximize a b [] = a
    maximize a b (move:moves)
        | a' >= b  = a'
        | otherwise = maximize a' b moves
      where
        a' = max a (minimax (applyMove board player move) (opposingPlayer player) (depth - 1) a b)

    minimize :: Int -> Int -> [[Coordinate]] -> Int
    minimize a b [] = b
    minimize a b (move:moves)
        | a >= b' = b'
        | otherwise = minimize a b' moves
      where
        b' = min b (minimax (applyMove board player move) (opposingPlayer player) (depth - 1) a b)

-- Board evaluation
evaluateBoard :: Board -> Player -> Int
evaluateBoard board player = scoreForPlayer - scoreForOpponent
  where
    scoreForPlayer = length [pos | (pos, state) <- assocs (grid board), state == Placed player]
    scoreForOpponent = length [pos | (pos, state) <- assocs (grid board), state == Placed (opposingPlayer player)]
```

othello/app/side_side_3.hs

```haskell
{-# LANGUAGE BangPatterns #-}

import System.Environment (getArgs)
import System.IO (readFile)
import Data.Array (array, listArray, assocs)
import Data.Foldable (maximumBy)
import Data.Ord (comparing)
import Control.Parallel.Strategies (parMap,parListChunk, rdeepseq)
import Othello.GameLogic (
    GameSetup(..), Player(..), initialGameState, Board(..), DiscState(..),
    movesForPlayer, applyMove, Coordinate, opposingPlayer
    )
import GHC.Conc (numCapabilities)
import Data.List.Split (chunksOf)

-- Parse a custom board state from a string
parseCustomBoard :: String -> Int -> Board
parseCustomBoard input dim = Board {
    grid = listArray ((0, 0), (dim-1, dim-1)) discStates,
    boardDim = dim
  }
  where
    rows = lines input
    discStates = concatMap parseRow rows
    parseRow row = map parseDiscState (words row)
    parseDiscState "E" = Empty
    parseDiscState "R" = Placed Red
    parseDiscState "B" = Placed Blue
    parseDiscState _   = error "Invalid disc state in custom board. Use 'E', 'R', or 'B'."


-- Define a simple entry point for the program
main :: IO ()
main = do
    -- Get command-line arguments
    args <- getArgs

    -- Parse depth arguments
    let (depth, parallelDepth, inputFile) = case args of
            (d:p:file:_) -> (read d, read p, Just file)
            (d:p:_)      -> (read d, read p, Nothing)
            _            -> error "Usage: <depth> <parallelDepth> [gameboard file]"

    -- Load the custom board or use the default initial game state
    gameSetup <- case inputFile of
        Just file -> do
            content <- readFile file
```

```haskell
            let customBoard = parseCustomBoard content 8 -- Assume an 8x8 board for simplicity
            return GameSetup {
                board = customBoard,
                aiPlays = [Red, Blue],
                searchDepth = depth
            }
        Nothing -> return (initialGameState 8 [Red, Blue] depth)

    putStrLn $ "Using minimax depth: " ++ show depth
    putStrLn $ "Parallelizing at depth: " ++ show parallelDepth

    let currentPlayer = Red

    let moves = getAIMove gameSetup currentPlayer depth parallelDepth
    if null moves
        then putStrLn "No valid moves available."
        else do
            let selectedMove = head moves
            let updatedBoard = applyMove (board gameSetup) currentPlayer [selectedMove]
            putStrLn $ "Next move: " ++ show selectedMove
            putStrLn "Updated board state:"
            print updatedBoard

-- Define a custom AI move function using alpha-beta pruning
getAIMove :: GameSetup -> Player -> Int -> Int -> [Coordinate]
getAIMove setup player depth parallelDepth
    | null possibleMoves = [] -- No moves available
    | otherwise = fst $ maximumBy (comparing snd) evaluatedMoves
  where
    possibleMoves = movesForPlayer (board setup) player

    evaluatedMoves =
        if depth == parallelDepth
        then parallelEvaluate possibleMoves
        else sequentialEvaluate possibleMoves

    sequentialEvaluate moves =
        [(move, minimax (applyMove (board setup) player move) (opposingPlayer player) (depth - 1) parallelDepth minBound maxBound)
        | move <- moves]

    parallelEvaluate moves =
        let chunkSize = max 1 (length moves `div` numCapabilities) -- Split moves into chunks
            eval move = (move, minimax (applyMove (board setup) player move) (opposingPlayer player) (depth - 1) parallelDepth minBound maxBound)
        in concat $ parMap rdeepseq (map eval) (chunksOf chunkSize moves)

-- Minimax algorithm with alpha-beta pruning
minimax :: Board -> Player -> Int -> Int -> Int -> Int -> Int
```

```haskell
minimax board player depth parallelDepth alpha beta
    | depth == 0 || null possibleMoves = evaluateBoard board player
    | player == maximizingPlayer = maximize alpha beta possibleMoves
    | otherwise = minimize alpha beta possibleMoves
  where
    possibleMoves = movesForPlayer board player
    maximizingPlayer = Red

    maximize :: Int -> Int -> [[Coordinate]] -> Int
    maximize a b [] = a
    maximize a b (move:moves)
        | a' >= b  = a'
        | otherwise = maximize a' b moves
      where
        a' = max a (nextEval move a b)

    minimize :: Int -> Int -> [[Coordinate]] -> Int
    minimize a b [] = b
    minimize a b (move:moves)
        | a >= b' = b'
        | otherwise = minimize a b' moves
      where
        b' = min b (nextEval move a b)

    nextEval move a b
        | depth == parallelDepth =
            let results = parMap rdeepseq eval possibleMoves
            in if player == maximizingPlayer then maximum results else minimum results
        | otherwise =
            minimax (applyMove board player move) (opposingPlayer player) (depth - 1) parallelDepth a b
        where
            eval move = minimax (applyMove board player move) (opposingPlayer player) (depth - 1) parallelDepth a b



-- Board evaluation
evaluateBoard :: Board -> Player -> Int
evaluateBoard board player = scoreForPlayer - scoreForOpponent
  where
    scoreForPlayer = length [pos | (pos, state) <- assocs (grid board), state == Placed player]
    scoreForOpponent = length [pos | (pos, state) <- assocs (grid board), state == Placed (opposingPlayer player)]



```

othello/app/side_side_4.hs

```haskell
{-# LANGUAGE BangPatterns #-}

import System.Environment (getArgs)
import System.IO (readFile)
import Data.Array (array, listArray, assocs)
import Data.Foldable (maximumBy)
import Data.Ord (comparing)
import Control.Parallel.Strategies (parMap,parListChunk, rdeepseq)
import Othello.GameLogic (
    GameSetup(..), Player(..), initialGameState, Board(..), DiscState(..),
    movesForPlayer, applyMove, Coordinate, opposingPlayer
    )
import GHC.Conc (numCapabilities)
import Data.List.Split (chunksOf)

-- Parse a custom board state from a string
parseCustomBoard :: String -> Int -> Board
parseCustomBoard input dim = Board {
    grid = listArray ((0, 0), (dim-1, dim-1)) discStates,
    boardDim = dim
  }
  where
    rows = lines input
    discStates = concatMap parseRow rows
    parseRow row = map parseDiscState (words row)
    parseDiscState "E" = Empty
    parseDiscState "R" = Placed Red
    parseDiscState "B" = Placed Blue
    parseDiscState _   = error "Invalid disc state in custom board. Use 'E', 'R', or 
'B'."

-- Main entry point
-- Define a simple entry point for the program
main :: IO ()
main = do
    -- Get command-line arguments
    args <- getArgs

    -- Parse depth arguments
    let (depth, parallelDepth, inputFile) = case args of
            (d:p:file:_) -> (read d, read p, Just file)
            (d:p:_)      -> (read d, read p, Nothing)
            _            -> error "Usage: <depth> <parallelDepth> [gameboard file]"

    -- Load the custom board or use the default initial game state
    gameSetup <- case inputFile of
        Just file -> do
            content <- readFile file
            let customBoard = parseCustomBoard content 8 -- Assume an 8x8 board for 
simplicity
```

```haskell
      return GameSetup {
        board = customBoard,
        aiPlays = [Red, Blue],
        searchDepth = depth
      }
    Nothing -> return (initialGameState 8 [Red, Blue] depth)

  putStrLn $ "Using minimax depth: " ++ show depth
  putStrLn $ "Parallelizing at depth: " ++ show parallelDepth

  let currentPlayer = Red

  let moves = getAIMove gameSetup currentPlayer depth parallelDepth
  if null moves
    then putStrLn "No valid moves available."
    else do
      let selectedMove = head moves
      let updatedBoard = applyMove (board gameSetup) currentPlayer [selectedMove]
      putStrLn $ "Next move: " ++ show selectedMove
      putStrLn "Updated board state:"
      print updatedBoard


getAIMove :: GameSetup -> Player -> Int -> Int -> [Coordinate]
getAIMove setup player depth parallelDepth
  | null possibleMoves = [] -- No moves available
  | otherwise = fst $ maximumBy (comparing snd) evaluatedMoves
 where
  possibleMoves = movesForPlayer (board setup) player

  evaluatedMoves =
    if depth == parallelDepth
    then parallelEvaluate possibleMoves
    else sequentialEvaluate possibleMoves

  sequentialEvaluate moves =
    [(move, minimax (applyMove (board setup) player move) (opposingPlayer player) (depth - 1) parallelDepth minBound maxBound)
    | move <- moves]

  parallelEvaluate moves =
    let chunkSize = max 1 (length moves `div` numCapabilities) -- Split moves into chunks
      eval move = (move, minimax (applyMove (board setup) player move) (opposingPlayer player) (depth - 1) parallelDepth minBound maxBound)
    in concat $ parMap rdeepseq (map eval) (chunksOf chunkSize moves)

-- Minimax algorithm with alpha-beta pruning
minimax :: Board -> Player -> Int -> Int -> Int -> Int -> Int
minimax board player depth parallelDepth alpha beta
  | depth == 0 || null possibleMoves = evaluateBoard board player --Base case
```

```haskell
    | player == maximizingPlayer = maximize alpha beta possibleMoves
    | otherwise = minimize alpha beta possibleMoves
  where
    possibleMoves = movesForPlayer board player
    maximizingPlayer = Red

    maximize :: Int -> Int -> [[Coordinate]] -> Int
    maximize a b [] = a
    maximize a b (move:moves)
        | a' >= b  = a'
        | otherwise = maximize a' b moves
      where
        a' = max a (nextEval move a b)

    minimize :: Int -> Int -> [[Coordinate]] -> Int
    minimize a b [] = b
    minimize a b (move:moves)
        | a >= b' = b'
        | otherwise = minimize a b' moves
      where
        b' = min b (nextEval move a b)

    nextEval move a b
        | depth == parallelDepth =
            let results = parMap rdeepseq eval possibleMoves
            in if player == maximizingPlayer then maximum results else minimum results
        | otherwise =
            minimax (applyMove board player move) (opposingPlayer player) (depth - 1) parallelDepth a b
        where
            eval move = minimax (applyMove board player move) (opposingPlayer player) (depth - 1) parallelDepth a b



-- Board evaluation
evaluateBoard :: Board -> Player -> Int
evaluateBoard board player = scoreForPlayer - scoreForOpponent
  where
    scoreForPlayer = length [pos | (pos, state) <- assocs (grid board), state == Placed player]
    scoreForOpponent = length [pos | (pos, state) <- assocs (grid board), state == Placed (opposingPlayer player)]
```

othello/app/side_side_5.hs

```haskell
module Main where

import GHC.Conc (numCapabilities)
```

```haskell
import System.Environment (getArgs)
import System.IO (readFile)
import Data.Array (array, listArray, assocs)
import Data.Foldable (maximumBy)
import Data.Ord (comparing)
import Control.Parallel (par, pseq)
import Control.Parallel.Strategies (parList, parListChunk, rdeepseq, using, parMap)
import Othello.GameLogic (
    GameSetup(..), Player(..), initialGameState, Board(..), DiscState(..),
    movesForPlayer, applyMove, Coordinate, opposingPlayer
  )

-- Parse a custom board state from a string
parseCustomBoard :: String -> Int -> Board
parseCustomBoard input dim = Board {
    grid = listArray ((0, 0), (dim-1, dim-1)) discStates,
    boardDim = dim
  }
  where
    rows = lines input
    discStates = concatMap parseRow rows
    parseRow row = map parseDiscState (words row)
    parseDiscState "E" = Empty
    parseDiscState "R" = Placed Red
    parseDiscState "B" = Placed Blue
    parseDiscState _   = error "Invalid disc state in custom board. Use 'E', 'R', or 'B'."

-- Define a simple entry point for the program
main :: IO ()
main = do
    -- Get command-line arguments
    args <- getArgs

    -- Parse depth arguments
    let (depth, parallelDepth, inputFile) = case args of
            (d:p:file:_) -> (read d, read p, Just file)
            (d:p:_)      -> (read d, read p, Nothing)
            _            -> error "Usage: <depth> <parallelDepth> [gameboard file]"

    -- Load the custom board or use the default initial game state
    gameSetup <- case inputFile of
        Just file -> do
            content <- readFile file
            let customBoard = parseCustomBoard content 8 -- Assume an 8x8 board for simplicity
            return GameSetup {
                board = customBoard,
                aiPlays = [Red, Blue],
                searchDepth = depth
```

```haskell
            }
        Nothing -> return (initialGameState 8 [Red, Blue] depth)

    putStrLn $ "Using minimax depth: " ++ show depth
    putStrLn $ "Parallelizing at depth: " ++ show parallelDepth

    let currentPlayer = Red

    -- Calculate the next move for the current player with the specified depth
    let moves = getAIMove gameSetup currentPlayer depth parallelDepth
    if null moves
        then putStrLn "No valid moves available."
        else do
            let selectedMove = head moves
            let updatedBoard = applyMove (board gameSetup) currentPlayer [selectedMove]
            putStrLn $ "Next move: " ++ show selectedMove
            putStrLn "Updated board state:"
            print updatedBoard

-- Minimax with optional parallelization
minimax :: Board -> Player -> Int -> Int -> Int -> Int -> Int
minimax board player depth alpha beta parallelDepth
    | depth == 0 || null possibleMoves = evaluateBoard board player -- Base case
    | depth == parallelDepth = parallelMinimaxAggressive board player depth alpha beta possibleMoves
    | otherwise = sequentialMinimax possibleMoves alpha beta
  where
    possibleMoves = movesForPlayer board player
    maximizingPlayer = Red -- Assume Red is the maximizing player

    -- Sequential minimax
    sequentialMinimax :: [[Coordinate]] -> Int -> Int -> Int
    sequentialMinimax [] a _ = a -- No more moves, return alpha
    sequentialMinimax (move:moves) a b
        | a >= b    = a -- Prune the rest of the tree
        | otherwise = sequentialMinimax moves a' b
      where
        a' = max a (evaluateMove board player (depth - 1) a b move)

    evaluateMove :: Board -> Player -> Int -> Int -> Int -> [Coordinate] -> Int
    evaluateMove b p d a b' move = minimax (applyMove b p move) (opposingPlayer p) d a b' parallelDepth

-- parallelMinimax
parallelMinimaxAggressive :: Board -> Player -> Int -> Int -> Int -> [[Coordinate]] -> Int
parallelMinimaxAggressive board player depth a b moves =
    let chunkedMoves = map (:[]) moves `using` parListChunk (max 1 (length moves `div` (round( 1 * fromIntegral numCapabilities)))) rdeepseq
        results = map (maximum . map (evaluateMove board player (depth - 1) a b)) chunkedMoves
```

```haskell
    in maximum results
  where
    evaluateMove :: Board -> Player -> Int -> Int -> Int -> [Coordinate] -> Int
    evaluateMove b p d a b' move = minimax (applyMove b p move) (opposingPlayer p) d a b' depth

-- AI
getAIMove :: GameSetup -> Player -> Int -> Int -> [Coordinate]
getAIMove setup player depth parallelDepth
    | null possibleMoves = [] -- No moves available
    | otherwise = fst $ maximumBy (comparing snd) evaluatedMoves
  where
    -- Get all possible moves for the current player
    possibleMoves = movesForPlayer (board setup) player

    -- Evaluate each move using minimax
    evaluatedMoves = [
        (move, minimax (applyMove (board setup) player move) (opposingPlayer player) (depth - 1) minBound maxBound parallelDepth)
        | move <- possibleMoves
      ]

-- Evaluate Board
evaluateBoard :: Board -> Player -> Int
evaluateBoard board player = scoreForPlayer - scoreForOpponent
  where
    scoreForPlayer = length [pos | (pos, state) <- assocs (grid board), state == Placed player]
    scoreForOpponent = length [pos | (pos, state) <- assocs (grid board), state == Placed (opposingPlayer player)]

```

othello/app/side_side_6.hs

```haskell
module Main where

import System.Environment (getArgs)
import System.IO (readFile)
import Data.Array (array, listArray, assocs)
import Control.Parallel.Strategies (parMap, rdeepseq, parBuffer, using)
import Data.List (maximumBy)
import Data.Ord (comparing)
import Othello.GameLogic (
    GameSetup(..), Player(..), initialGameState, Board(..), DiscState(..),
    movesForPlayer, applyMove, Coordinate, opposingPlayer
    )

-- Parse a custom board state from a string
parseCustomBoard :: String -> Int -> Board
parseCustomBoard input dim = Board {
    grid = listArray ((0, 0), (dim-1, dim-1)) discStates,
```

```haskell
    boardDim = dim
  }
  where
    rows = lines input
    discStates = concatMap parseRow rows
    parseRow row = map parseDiscState (words row)
    parseDiscState "E" = Empty
    parseDiscState "R" = Placed Red
    parseDiscState "B" = Placed Blue
    parseDiscState _   = error "Invalid disc state in custom board. Use 'E', 'R', or 'B'."

-- Define a simple entry point for the program
main :: IO ()
main = do
    -- Get command-line arguments
    args <- getArgs

    -- Parse depth arguments
    let (depth, parallelDepth, inputFile) = case args of
            (d:p:file:_) -> (read d, read p, Just file)
            (d:p:_)      -> (read d, read p, Nothing)
            _            -> error "Usage: <depth> <parallelDepth> [gameboard file]"

    -- Load the custom board or use the default initial game state
    gameSetup <- case inputFile of
        Just file -> do
            content <- readFile file
            let customBoard = parseCustomBoard content 8 -- Assume an 8x8 board for simplicity
            return GameSetup {
                board = customBoard,
                aiPlays = [Red, Blue],
                searchDepth = depth
            }
        Nothing -> return (initialGameState 8 [Red, Blue] depth)

    putStrLn $ "Using minimax depth: " ++ show depth
    putStrLn $ "Parallelizing at depth: " ++ show parallelDepth

    let currentPlayer = Red

    -- Calculate the next move for the current player with the specified depth
    let moves = getAIMoveParallel gameSetup currentPlayer depth parallelDepth
    if null moves
        then putStrLn "No valid moves available."
        else do
            let selectedMove = head moves
            let updatedBoard = applyMove (board gameSetup) currentPlayer [selectedMove]
            putStrLn $ "Next move: " ++ show selectedMove
```

```haskell
            putStrLn "Updated board state:"
            print updatedBoard

-- Parallel Minimax with Alpha-Beta Pruning and controlled parallel depth
minimax :: Board -> Player -> Int -> Int -> Int -> Int -> Int
minimax board player depth alpha beta parallelDepth
    | depth == 0 || null possibleMoves = evaluateBoard board player -- Base case
    | depth > parallelDepth = sequentialMinimax alpha beta possibleMoves
    | depth == parallelDepth = parallelMinimax alpha beta possibleMoves
    | otherwise = sequentialMinimax alpha beta possibleMoves
  where
    possibleMoves = movesForPlayer board player

    -- Sequential minimax
    sequentialMinimax :: Int -> Int -> [[Coordinate]] -> Int
    sequentialMinimax a b [] = if player == maximizingPlayer then a else b
    sequentialMinimax a b (move:moves)
        | player == maximizingPlayer = maximizing a b moves move
        | otherwise = minimizing a b moves move
      where
        maximizing a b moves move = sequentialMinimax (max a (minimax (applyMove board player move) (opposingPlayer player) (depth - 1) a b parallelDepth)) b moves
        minimizing a b moves move = sequentialMinimax a (min b (minimax (applyMove board player move) (opposingPlayer player) (depth - 1) a b parallelDepth)) moves

    -- Parallel minimax
    parallelMinimax :: Int -> Int -> [[Coordinate]] -> Int
    parallelMinimax a b moves =
        let results = map evaluateMove moves `using` parBuffer 2 rdeepseq
        in if player == maximizingPlayer
           then maximum results
           else minimum results
      where
        evaluateMove move = minimax (applyMove board player move) (opposingPlayer player) (depth - 1) a b parallelDepth

    maximizingPlayer = Red -- Assume Red is the maximizing player

-- AI with parallelism and controlled depth
getAIMoveParallel :: GameSetup -> Player -> Int -> Int -> [Coordinate]
getAIMoveParallel setup player depth parallelDepth
    | null possibleMoves = [] -- No moves available
    | otherwise = fst $ maximumBy (comparing snd) evaluatedMoves
  where
    -- Get all possible moves for the current player
    possibleMoves = movesForPlayer (board setup) player

    -- Evaluate moves in parallel
    evaluatedMoves = map evaluateMove possibleMoves `using` parBuffer 2 rdeepseq
    evaluateMove move = (move, minimax (applyMove (board setup) player move) (opposingPlayer player) (depth - 1) minBound maxBound parallelDepth) `using` rdeepseq
```

```haskell

evaluateBoard :: Board -> Player -> Int
evaluateBoard board player = scoreForPlayer - scoreForOpponent
  where
    scoreForPlayer = length [pos | (pos, state) <- assocs (grid board), state == Placed
player]
    scoreForOpponent = length [pos | (pos, state) <- assocs (grid board), state ==
Placed (opposingPlayer player)]

```

othello/app/side_side_7.hs

```haskell
module Main where


import Data.List.Split (chunksOf)
import GHC.Conc (numCapabilities)
import System.Environment (getArgs)
import System.IO (readFile)
import Data.Array (array, listArray, assocs)
import Control.Parallel.Strategies (parMap, rdeepseq, parBuffer, using)
import Data.List (maximumBy)
import Data.Ord (comparing)
import Othello.GameLogic (
    GameSetup(..), Player(..), initialGameState, Board(..), DiscState(..),
    movesForPlayer, applyMove, Coordinate, opposingPlayer
    )

-- Parse a custom board state from a string
parseCustomBoard :: String -> Int -> Board
parseCustomBoard input dim = Board {
    grid = listArray ((0, 0), (dim-1, dim-1)) discStates,
    boardDim = dim
  }
  where
    rows = lines input
    discStates = concatMap parseRow rows
    parseRow row = map parseDiscState (words row)
    parseDiscState "E" = Empty
    parseDiscState "R" = Placed Red
    parseDiscState "B" = Placed Blue
    parseDiscState _   = error "Invalid disc state in custom board. Use 'E', 'R', or
'B'."

-- Define a simple entry point for the program
-- Define a simple entry point for the program
main :: IO ()
main = do
    -- Get command-line arguments
    args <- getArgs

```

```
    -- Parse depth arguments
    let (depth, parallelDepth, inputFile) = case args of
            (d:p:file:_) -> (read d, read p, Just file)
            (d:p:_)      -> (read d, read p, Nothing)
            _            -> error "Usage: <depth> <parallelDepth> [gameboard file]"

    -- Load the custom board or use the default initial game state
    gameSetup <- case inputFile of
        Just file -> do
            content <- readFile file
            let customBoard = parseCustomBoard content 8 -- Assume an 8x8 board for simplicity
            return GameSetup {
                board = customBoard,
                aiPlays = [Red, Blue],
                searchDepth = depth
            }
        Nothing -> return (initialGameState 8 [Red, Blue] depth)

    putStrLn $ "Using minimax depth: " ++ show depth
    putStrLn $ "Parallelizing at depth: " ++ show parallelDepth

    let currentPlayer = Red

    -- Calculate the next move for the current player with the specified depth
    let moves = getAIMoveParallel gameSetup currentPlayer depth parallelDepth
    if null moves
        then putStrLn "No valid moves available."
        else do
            let selectedMove = head moves
            let updatedBoard = applyMove (board gameSetup) currentPlayer [selectedMove]
            putStrLn $ "Next move: " ++ show selectedMove
            putStrLn "Updated board state:"
            print updatedBoard

--minimax search tree
minimax :: Board -> Player -> Int -> Int -> Int -> Int -> Int
minimax board player depth alpha beta parallelDepth
    | depth == 0 || null possibleMoves = evaluateBoard board player -- Base case
    | depth > parallelDepth = sequentialMinimax alpha beta possibleMoves
    | depth == parallelDepth = parallelMinimax alpha beta possibleMoves
    | otherwise = sequentialMinimax alpha beta possibleMoves
  where
    possibleMoves = movesForPlayer board player

    -- Sequential minimax
    sequentialMinimax :: Int -> Int -> [[Coordinate]] -> Int
    sequentialMinimax a b [] = if player == maximizingPlayer then a else b
    sequentialMinimax a b (move:moves)
        | player == maximizingPlayer = maximizing a b moves move
```

```haskell
        | otherwise = minimizing a b moves move
      where
        maximizing a b moves move = sequentialMinimax (max a (minimax (applyMove board player move) (opposingPlayer player) (depth - 1) a b parallelDepth)) b moves
        minimizing a b moves move = sequentialMinimax a (min b (minimax (applyMove board player move) (opposingPlayer player) (depth - 1) a b parallelDepth)) moves

    -- Parallel minimax
    parallelMinimax :: Int -> Int -> [[Coordinate]] -> Int
    parallelMinimax a b moves =
        let
            -- Number of threads available
            threads = numCapabilities

            bufferSize = max 1 (round (2 * fromIntegral threads))
            chunkSize = max 1 (length moves `div` (max 1 (round (fromIntegral threads * fromIntegral threads/3))))
            chunkedMoves = chunksOf chunkSize moves

            results = map (maximum . map evaluateMove) chunkedMoves `using` parBuffer bufferSize rdeepseq
        in if player == maximizingPlayer
           then maximum results
           else minimum results
      where
        -- Evaluates a single move
        evaluateMove move = minimax (applyMove board player move) (opposingPlayer player) (depth - 1) a b parallelDepth
    maximizingPlayer = Red -- Assume Red is the maximizing player

-- AI with parallelism and controlled depth
getAIMoveParallel :: GameSetup -> Player -> Int -> Int -> [Coordinate]
getAIMoveParallel setup player depth parallelDepth
    | null possibleMoves = [] -- No moves available
    | otherwise = fst $ maximumBy (comparing snd) evaluatedMoves
  where
    -- Get all possible moves for the current player
    possibleMoves = movesForPlayer (board setup) player

    -- Evaluate moves in parallel
    evaluatedMoves = map evaluateMove possibleMoves `using` parBuffer 2 rdeepseq
    evaluateMove move = (move, minimax (applyMove (board setup) player move) (opposingPlayer player) (depth - 1) minBound maxBound parallelDepth) `using` rdeepseq

evaluateBoard :: Board -> Player -> Int
evaluateBoard board player = scoreForPlayer - scoreForOpponent
  where
    scoreForPlayer = length [pos | (pos, state) <- assocs (grid board), state == Placed player]
    scoreForOpponent = length [pos | (pos, state) <- assocs (grid board), state == Placed (opposingPlayer player)]
```

othello/app/side_side_8.hs

```haskell
module Main where

import System.Environment (getArgs)
import System.IO (readFile)
import Data.Array (array, listArray, assocs)
import Data.Foldable (maximumBy)
import Data.Ord (comparing)
import Control.Parallel (par, pseq)
import Control.Parallel.Strategies (parList, parListChunk, rdeepseq, using,parMap)
import Othello.GameLogic (
    GameSetup(..), Player(..), initialGameState, Board(..), DiscState(..),
    movesForPlayer, applyMove, Coordinate, opposingPlayer
  )

-- Parse a custom board state from a string
parseCustomBoard :: String -> Int -> Board
parseCustomBoard input dim = Board {
    grid = listArray ((0, 0), (dim-1, dim-1)) discStates,
    boardDim = dim
  }
  where
    rows = lines input
    discStates = concatMap parseRow rows
    parseRow row = map parseDiscState (words row)
    parseDiscState "E" = Empty
    parseDiscState "R" = Placed Red
    parseDiscState "B" = Placed Blue
    parseDiscState _   = error "Invalid disc state in custom board. Use 'E', 'R', or 'B'."

-- Define a simple entry point for the program
main :: IO ()
main = do
    -- Get command-line arguments
    args <- getArgs

    -- Parse depth arguments
    let (depth, parallelDepth, inputFile) = case args of
            (d:p:file:_) -> (read d, read p, Just file)
            (d:p:_)      -> (read d, read p, Nothing)
            _            -> error "Usage: <depth> <parallelDepth> [gameboard file]"

    -- Load the custom board or use the default initial game state
    gameSetup <- case inputFile of
        Just file -> do
            content <- readFile file
```

```haskell
            let customBoard = parseCustomBoard content 8 -- Assume an 8x8 board for simplicity
            return GameSetup {
                board = customBoard,
                aiPlays = [Red, Blue],
                searchDepth = depth
            }
        Nothing -> return (initialGameState 8 [Red, Blue] depth)

    putStrLn $ "Using minimax depth: " ++ show depth
    putStrLn $ "Parallelizing at depth: " ++ show parallelDepth

    let currentPlayer = Red

    -- Calculate the next move for the current player with the specified depth
    let moves = getAIMove gameSetup currentPlayer depth parallelDepth
    if null moves
        then putStrLn "No valid moves available."
        else do
            let selectedMove = head moves
            let updatedBoard = applyMove (board gameSetup) currentPlayer [selectedMove]
            putStrLn $ "Next move: " ++ show selectedMove
            putStrLn "Updated board state:"
            print updatedBoard

-- Minimax with optional parallelization
minimax :: Board -> Player -> Int -> Int -> Int -> Int
minimax board player depth alpha beta parallelDepth
    | depth == 0 || null possibleMoves = evaluateBoard board player -- Base case: evaluate board
    | depth == parallelDepth = parallelMinimax possibleMoves alpha beta
    | otherwise = sequentialMinimax possibleMoves alpha beta
  where
    possibleMoves = movesForPlayer board player
    maximizingPlayer = Red -- Assume Red is the maximizing player

    -- Parallel minimax evaluation (Option 1: Parallelize at one depth)
    parallelMinimax :: [[Coordinate]] -> Int -> Int -> Int
    parallelMinimax moves a b =
        let results = parMap rdeepseq (evaluateMove board player (depth - 1) a b) moves
        in maximum results

    -- Sequential minimax evaluation
    sequentialMinimax :: [[Coordinate]] -> Int -> Int -> Int
    sequentialMinimax [] a _ = a -- No more moves, return alpha
    sequentialMinimax (move:moves) a b
        | a >= b    = a -- Prune the rest of the tree
        | otherwise = sequentialMinimax moves a' b
      where
        a' = max a (evaluateMove board player (depth - 1) a b move)
```

```haskell

    evaluateMove :: Board -> Player -> Int -> Int -> Int -> [Coordinate] -> Int
    evaluateMove b p d a b' move = minimax (applyMove b p move) (opposingPlayer p) d a b' parallelDepth

-- Parallelization Option 2: Divide the Tree Aggressively
parallelMinimaxAggressive :: Board -> Player -> Int -> Int -> Int -> [[Coordinate]] -> Int
parallelMinimaxAggressive board player depth a b moves =
    let chunkedMoves = map (:[]) moves `using` parListChunk 2 rdeepseq
        results = map (maximum . map (evaluateMove board player (depth - 1) a b)) chunkedMoves
    in maximum results
  where
    evaluateMove :: Board -> Player -> Int -> Int -> Int -> [Coordinate] -> Int
    evaluateMove b p d a b' move = minimax (applyMove b p move) (opposingPlayer p) d a b' depth

-- AI
getAIMove :: GameSetup -> Player -> Int -> Int -> [Coordinate]
getAIMove setup player depth parallelDepth
    | null possibleMoves = [] -- No moves available
    | otherwise = fst $ maximumBy (comparing snd) evaluatedMoves
  where
    -- Get all possible moves for the current player
    possibleMoves = movesForPlayer (board setup) player

    -- Evaluate each move using minimax
    evaluatedMoves = [
        (move, minimax (applyMove (board setup) player move) (opposingPlayer player) (depth - 1) minBound maxBound parallelDepth)
        | move <- possibleMoves
      ]

-- Example board evaluation function
evaluateBoard :: Board -> Player -> Int
evaluateBoard board player = scoreForPlayer - scoreForOpponent
  where
    scoreForPlayer = length [pos | (pos, state) <- assocs (grid board), state == Placed player]
    scoreForOpponent = length [pos | (pos, state) <- assocs (grid board), state == Placed (opposingPlayer player)]

```

othello/app/side_side_9.hs

```haskell
module Main where

import GHC.Conc (numCapabilities)
import System.Environment (getArgs)
import System.IO (readFile)
```

```haskell
import Data.Array (array, listArray, assocs)
import Data.Foldable (maximumBy)
import Data.Ord (comparing)
import Control.Parallel (par, pseq)
import Control.Parallel.Strategies (parList, parListChunk, rdeepseq, using,parMap)
import Othello.GameLogic (
    GameSetup(..), Player(..), initialGameState, Board(..), DiscState(..),
    movesForPlayer, applyMove, Coordinate, opposingPlayer
  )

-- Parse a custom board state from a string
parseCustomBoard :: String -> Int -> Board
parseCustomBoard input dim = Board {
    grid = listArray ((0, 0), (dim-1, dim-1)) discStates,
    boardDim = dim
  }
  where
    rows = lines input
    discStates = concatMap parseRow rows
    parseRow row = map parseDiscState (words row)
    parseDiscState "E" = Empty
    parseDiscState "R" = Placed Red
    parseDiscState "B" = Placed Blue
    parseDiscState _   = error "Invalid disc state in custom board. Use 'E', 'R', or 'B'."

-- Define a simple entry point for the program
main :: IO ()
main = do
    -- Get command-line arguments
    args <- getArgs

    -- Parse depth arguments
    let (depth, parallelDepth, inputFile) = case args of
            (d:p:file:_) -> (read d, read p, Just file)
            (d:p:_)      -> (read d, read p, Nothing)
            _            -> error "Usage: <depth> <parallelDepth> [gameboard file]"

    -- Load the custom board or use the default initial game state
    gameSetup <- case inputFile of
        Just file -> do
            content <- readFile file
            let customBoard = parseCustomBoard content 8 -- Assume an 8x8 board for simplicity
            return GameSetup {
                board = customBoard,
                aiPlays = [Red, Blue],
                searchDepth = depth
            }
        Nothing -> return (initialGameState 8 [Red, Blue] depth)
```

```haskell

    putStrLn $ "Using minimax depth: " ++ show depth
    putStrLn $ "Parallelizing at depth: " ++ show parallelDepth

    let currentPlayer = Red

    -- Calculate the next move for the current player with the specified depth
    let moves = getAIMove gameSetup currentPlayer depth parallelDepth
    if null moves
        then putStrLn "No valid moves available."
        else do
            let selectedMove = head moves
            let updatedBoard = applyMove (board gameSetup) currentPlayer [selectedMove]
            putStrLn $ "Next move: " ++ show selectedMove
            putStrLn "Updated board state:"
            print updatedBoard

-- Minimax game board
minimax :: Board -> Player -> Int -> Int -> Int -> Int -> Int
minimax board player depth alpha beta parallelDepth
    | depth == 0 || null possibleMoves = evaluateBoard board player -- Base case
    | depth == parallelDepth = parallelMinimax possibleMoves alpha beta
    | otherwise = sequentialMinimax possibleMoves alpha beta
  where
    possibleMoves = movesForPlayer board player
    maximizingPlayer = Red -- Assume Red is the maximizing player

    -- Parallel minimax evaluation
    parallelMinimax :: [[Coordinate]] -> Int -> Int -> Int
    parallelMinimax moves a b =
        let results = parMap rdeepseq (evaluateMove board player (depth - 1) a b) moves
        in maximum results

    -- Sequential minimax evaluation
    sequentialMinimax :: [[Coordinate]] -> Int -> Int -> Int
    sequentialMinimax [] a _ = a -- No more moves, return alpha
    sequentialMinimax (move:moves) a b
        | a >= b    = a -- Prune the rest of the tree
        | otherwise = sequentialMinimax moves a' b
      where
        a' = max a (evaluateMove board player (depth - 1) a b move)

    evaluateMove :: Board -> Player -> Int -> Int -> Int -> [Coordinate] -> Int
    evaluateMove b p d a b' move = minimax (applyMove b p move) (opposingPlayer p) d a
b' parallelDepth

-- AI
getAIMove :: GameSetup -> Player -> Int -> Int -> [Coordinate]
getAIMove setup player depth parallelDepth
    | null possibleMoves = [] -- No moves available
```

```haskell
    | otherwise = fst $ maximumBy (comparing snd) evaluatedMoves
  where
    -- Get all possible moves for the current player
    possibleMoves = movesForPlayer (board setup) player

    -- Evaluate each move using minimax
    evaluatedMoves = [
        (move, minimax (applyMove (board setup) player move) (opposingPlayer player) (depth - 1) minBound maxBound parallelDepth)
        | move <- possibleMoves
      ]
--evaluate Board
evaluateBoard :: Board -> Player -> Int
evaluateBoard board player = scoreForPlayer - scoreForOpponent
  where
    scoreForPlayer = length [pos | (pos, state) <- assocs (grid board), state == Placed player]
    scoreForOpponent = length [pos | (pos, state) <- assocs (grid board), state == Placed (opposingPlayer player)]

```

othello/benchmark_performance.py

```python
import subprocess
import sys
import re
import argparse

def parse_output(stdout):
    total_time_pattern = r"Total\s+time\s+\d+\.\d+s\s+\(\s+([\d\.]+)s"
    tasks_pattern = r"TASKS:\s+(\d+)"
    sparks_pattern = r"SPARKS:\s+(\d+)"

    total_time_match = re.search(total_time_pattern, stdout)
    tasks_match = re.search(tasks_pattern, stdout)
    sparks_match = re.search(sparks_pattern, stdout)

    total_time = float(total_time_match.group(1)) if total_time_match else None
    tasks = int(tasks_match.group(1)) if tasks_match else None
    sparks = int(sparks_match.group(1)) if sparks_match else None

    return total_time, tasks, sparks

def run_benchmark(test_file="custom_game_1.txt", iterate="parallel_depth", depth=7, parallel_depth=1):
    results = {}
    iteration_range = range(1, max(depth, parallel_depth) + 1)
    threads_range = range(1, 9)

    if iterate == "depth":
        depth_range = iteration_range
```

```python
        parallel_depth_range = [parallel_depth]
        threads_range = [4]
    elif iterate == "parallel_depth":
        depth_range = [depth]
        parallel_depth_range = iteration_range
        threads_range = [4]
    elif iterate == "num_threads":
        depth_range = [depth]
        parallel_depth_range = [parallel_depth]
    else:
        print(f"Invalid iterate value: {iterate}. Use 'depth', 'parallel_depth', or 'num_threads'.")
        return

    for d in depth_range:
        for pd in parallel_depth_range:
            for threads in threads_range:
                print(f"Running tests with depth {d}, parallel depth {pd}, threads {threads}...")

                total_times = []
                total_tasks = []
                total_sparks = []

                for _ in range(5):
                    # Prepare the command
                    command = [
                        "cabal", "run", "othello", "--", str(d), str(pd), test_file, "+RTS", "-s", f"-N{threads}"
                    ]

                    try:
                        result = subprocess.run(command, capture_output=True, text=True, check=True)
                        stdout = result.stdout
                        stderr = result.stderr

                        total_time, tasks, sparks = parse_output(stdout + stderr)

                        if total_time is not None and tasks is not None and sparks is not None:
                            total_times.append(total_time)
                            total_tasks.append(tasks)
                            total_sparks.append(sparks)
                        else:
                            print(f"Warning: Missing data in the output for depth {d}, parallel depth {pd}, threads {threads}")


                    except subprocess.CalledProcessError as e:
                        print(f"Error running command {command}: {e}")
```

```
                        continue

                # Compute averages for this configuration
                if total_times:
                    avg_total_time = sum(total_times) / len(total_times)
                    avg_tasks = sum(total_tasks) / len(total_tasks)
                    avg_sparks = sum(total_sparks) / len(total_sparks)
                    results[(d, pd, threads)] = {
                        "avg_total_time": avg_total_time,
                        "avg_tasks": avg_tasks,
                        "avg_sparks": avg_sparks
                    }

    for (d, pd, threads), data in results.items():
        print(f"Depth {d}, Parallel depth {pd}, Threads {threads}:")
        print(f"  Average Total Time: {data['avg_total_time']:.3f}s")
        print(f"  Average Tasks: {data['avg_tasks']}")
        print(f"  Average Sparks: {data['avg_sparks']}")

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Run benchmarks for Othello.")
    parser.add_argument("--test_file", type=str, default="custom_game_1.txt", help="Path to the test file.")
    parser.add_argument("--iterate", type=str, choices=["depth", "parallel_depth", "num_threads"], default="parallel_depth",
                        help="Parameter to iterate over: 'depth', 'parallel_depth', or 'num_threads'.")
    parser.add_argument("--depth", type=int, default=7, help="Initial depth value.")
    parser.add_argument("--parallel_depth", type=int, default=1, help="Initial parallel depth value.")

    args = parser.parse_args()

    run_benchmark(
        test_file=args.test_file,
        iterate=args.iterate,
        depth=args.depth,
        parallel_depth=args.parallel_depth
    )

```

othello/cabal.project

```
packages: .

profiling: True
```

othello/custom_game_1.txt

```
E E E E E E E E
E E E E E E E E
E E E E E E E E
E E E R B E E E
E E E B R E E E
E E E E E E E E
E E E E E E E E
E E E E E E E E
```

othello/custom_game_2.txt

```
E E E E E E E E
E E E E E E E E
E E E E E E E E
E E E R B E E E
E E R B R B E E
E E E B R B E E
E E E E B E E E
E E E E E E E E
```

othello/haskell-othello.cabal

```
cabal-version:      2.4
name:               haskell-othello
version:            0.1.0.0

-- A short (one-line) description of the package.
synopsis:           Experiments in parallelizing an Othello minmax agent.

-- A longer description of the package.
description:        Experiments in parallelizing an Othello minmax agent.

-- A URL where users can report bugs.
bug-reports:        https://github.com/NoamHirschorn/PFP_final/issues

-- The license under which the package is released.
--license:            MIT
--license-file:       LICENSE

-- The package author(s).
--author:             Noam Hirschorn, Dan Ivanovich. Adapted from code by Arttu. Y

-- An email address to which users can send suggestions, bug reports, and patches.
--maintainer:         dmi2115@columbia.edu

-- A copyright notice.
copyright:          2024 Noam Hirschorn, Dan Ivanovich, Arttu. Y
category:           Games
```

```
-- extra-source-files: CHANGELOG.md

library
    exposed-modules:  Othello.GameLogic

    -- Modules included in this library but not exported.
    -- other-modules:

    -- LANGUAGE extensions used by modules in this package.
    -- other-extensions:

    build-depends:
        base >=4.14.3.0 && <4.19,
        array,
        parallel,
        deepseq >= 1.4,
        split

    hs-source-dirs:   src
    default-language: Haskell2010

executable othello
    main-is:          side_side_9.hs

    -- Modules included in this executable, other than Main.
    -- other-modules:

    -- LANGUAGE extensions used by modules in this package.
    -- other-extensions:

    build-depends:
        base >=4.14.3.0 && <4.19,
        array,
        parallel,
        deepseq >= 1.4,
        split,
        haskell-othello

    hs-source-dirs:   app
    default-language: Haskell2010

    ghc-options:      -threaded -rtsopts -with-rtsopts=-N -debug

```

othello/src/Othello/GameLogic.hs

```
{-#LANGUAGE InstanceSigs#-} -- Permit type declarations in instance definitions
{-# LANGUAGE TupleSections #-} -- Partial tuple constructors as functions
{-# LANGUAGE NamedFieldPuns #-} -- Allow more elegant construction of data
```

```haskell

module Othello.GameLogic where
import Data.Array ( Array, array, elems, inRange, bounds, (//) )
import qualified Data.Array ((!))
import Data.Foldable (maximumBy, minimumBy)
import Data.Ord (comparing)
import Data.Array (Array, assocs, elems, bounds, (//))
-- Score is an integer, and coordinate is a pair of integers
type UnitScore = Int
type Coordinate = (Int, Int)

-- A player is either Red, or Blue. Derive comparison and Show
data Player = Red | Blue deriving (Eq, Show)

-- Each spot on a board is either empty, or placed with some player
data DiscState = Empty | Placed Player deriving (Eq, Show)

-- Board is essentially a grid of disc states, with the size attached
data Board = Board {
    grid :: Array Coordinate DiscState,
    boardDim :: Int
} deriving (Show)

-- For purposes of Minimax AI, we will need to measure the score of a given state. It will be either a win, indeterminate with score of some kind (from the view of who is requesting it), or a loss
data BoardScore = Win | Indeterminate UnitScore | Lose deriving (Eq, Show)

-- Define a order for a board score. For least complexity, define ordering as a set of comparative properties between different scores
instance Ord BoardScore where
    (<=) :: BoardScore -> BoardScore -> Bool
    (<=) Lose _ = True -- Lose is the smallest and definitely equal
    (<=) (Indeterminate _) Lose = False -- Indeterminate is never less or equal to a win
    (<=) (Indeterminate _) Win = True -- Indeterminate is always less than a win
    (<=) (Indeterminate a) (Indeterminate b) = a <= b -- For two indeterminates, their respective ordering depends on their scores
    (<=) Win Win = True -- Win is equal with a win
    (<=) Win _ = False -- Otherwise, no

-- Define a scoring function; given a player and a board, what is their score?
score :: Player -> Board -> BoardScore
score player board
    | not (movesPossibleOnBoard board) && redLeading = if player == Red then Win else Lose
    | not (movesPossibleOnBoard board) && blueLeading = if player == Blue then Win else Lose
    | otherwise = Indeterminate (if player == Red then redCount else blueCount)
    where
        redLeading = redCount > blueCount
```

```haskell
        blueLeading = blueCount > redCount

        (redCount, blueCount) = pieceCount board

data GameSetup = GameSetup {
    board :: Board, -- Board
    aiPlays :: [Player], -- Which turns AI plays?
    searchDepth :: Int -- Search depth
} deriving (Show)
-- Core functions

-- Count of pieces, per color, for the board
pieceCount :: Board -> (Int, Int)
pieceCount board = foldr adder (0,0) (elems $ grid board) -- Add element by element, start with zero scores for both
    where
        adder :: DiscState -> (Int, Int) -> (Int, Int)
        adder state count@(red, blue) = case state of
            Empty -> count
            Placed Red -> (red+1, blue)
            Placed Blue -> (red, blue+1)

-- Definition of the opposing player for a given player
opposingPlayer :: Player -> Player
opposingPlayer Red = Blue
opposingPlayer Blue = Red

-- Calculating which player is winning by their score; this does not consider if there are more turns remaining
playerWithBestScore :: Board -> Maybe Player
playerWithBestScore board
    | red == blue = Nothing
    | otherwise = if red > blue then Just Red else Just Blue
    where
        (red, blue) = pieceCount board

-- Define an indexing operation for a board, quite alike what Arrays have
(!) :: Board -> Coordinate -> DiscState
board ! coordinate = (Data.Array.!) (grid board) coordinate

-- Define a validity check operator for indexes; this will return true if the index is acceptable
(!?) :: Board -> Coordinate -> Bool
board !? coord = inRange (bounds $ grid board) coord

-- Define a grid comprehension function; mapping over coordinates of a grid, construct some array of data
comprehensionByBoard :: Board -> (Coordinate -> x) -> [x]
comprehensionByBoard board = comprehensionByDim size
    where
        size = boardDim board
```

```haskell

comprehensionByDim :: Int -> (Coordinate -> x) -> [x]
comprehensionByDim size func = [ func (a,b) | a <- [0..size-1], b <- [0..size-1]]

initialGameState :: Int -> [Player] -> Int -> GameSetup
initialGameState dim aiPlays searchDepth = GameSetup { aiPlays, searchDepth, board }
    where
        board = Board { boardDim = dim, grid = array ((0, 0), (dim-1,dim-1)) (comprehensionByDim dim (\p -> (p, startPieces p))) }

        startPieces :: Coordinate -> DiscState
        startPieces (cx, cy)
            | cx == (dim `div` 2) - 1 && cy == (dim `div` 2) - 1 = Placed Blue
            | cx == (dim `div` 2) && cy == (dim `div` 2) = Placed Blue
            | cx == (dim `div` 2) - 1 && cy == (dim `div` 2) = Placed Red
            | cx == (dim `div` 2) && cy == (dim `div` 2) - 1 = Placed Red
            | otherwise = Empty
---- Moves and AI

-- For a given board, player and coordinate, determine what coordinates should be changed to player's color. If an empty list is returned, move is not valid
-- This also includes the starting point given
getMovesOnPoint :: Board -> Player -> Coordinate -> [Coordinate]
getMovesOnPoint board player startCoord@(sx, sy)
    | not (board !? startCoord) = [] -- Coordinate is not a valid position
    | startPiece /= Empty = [] -- Starting piece is not empty
    | null resultSteps = [] -- No valid steps exist
    | otherwise = startCoord : resultSteps
    where
        resultSteps = concatMap walkAndMark directions -- Valid steps are a concatenation of walked directions - per rules, we can and must mark all branched paths

        walkAndMark :: Coordinate -> [Coordinate] -- If stepping from a given direction, what can we mark (excluding start position)?
        walkAndMark dir@(dx, dy) = walkAndMark' (sx+dx, sy+dy) dir []

        walkAndMark' :: Coordinate -> Coordinate -> [Coordinate] -> [Coordinate] -- Current position, direction, found already
        walkAndMark' cur@(cx, cy) dir@(dx, dy) found
            | isEndPiece = found -- End piece, terminate here
            | isValidTraversalPiece = walkAndMark' (cx+dx, cy+dy) dir (cur:found) -- Can traverse, step forward
            | otherwise = [] -- No valid way to travel nor end, return nothing
            where
                isEndPiece = isValidPos && board ! cur == Placed player
                isValidTraversalPiece = isValidPos && board ! cur == Placed (opposingPlayer player)
                isValidPos = board !? cur

        directions = [(-1,-1), (-1,0), (-1,1), (0,-1), (0,1), (1,-1), (1,0), (1,1)] -- Which directions need to be checked. As defined in the rules, we work on either
```

```
horizontal, vertical or diagonal lines
        startPiece = board ! startCoord

movesForPlayer :: Board -> Player -> [[Coordinate]]  -- Return all movesets that are nonempty, for a given player?
movesForPlayer board player = filter (not . null) mappedCoords
    where
        mappedCoords = comprehensionByBoard board (getMovesOnPoint board player)

movesPossibleOnBoard :: Board -> Bool -- Are there any moves possible on board?
movesPossibleOnBoard board = not (null (movesForPlayer board Red) && null (movesForPlayer board Blue))

applyMove :: Board -> Player -> [Coordinate] -> Board
applyMove board player moveList = board { grid = grid board // map (, Placed player) moveList }


```

othello/src/Othello/backup_Gamelog.hs

```
{-#LANGUAGE InstanceSigs#-} -- Permit type declarations in instance definitions
{-# LANGUAGE TupleSections #-} -- Partial tuple constructors as functions
{-# LANGUAGE NamedFieldPuns #-} -- Allow more elegant construction of data

module Othello.GameLogic where
import Data.Array ( Array, array, elems, inRange, bounds, (//) )
import qualified Data.Array ((!))
import Data.Foldable (maximumBy, minimumBy)
import Data.Ord (comparing)
import Data.Array (Array, assocs, elems, bounds, (//))
-- Score is an integer, and coordinate is a pair of integers
type UnitScore = Int
type Coordinate = (Int, Int)

-- A player is either Red, or Blue. Derive comparison and Show
data Player = Red | Blue deriving (Eq, Show)

-- Each spot on a board is either empty, or placed with some player
data DiscState = Empty | Placed Player deriving (Eq, Show)

-- Board is essentially a grid of disc states, with the size attached
data Board = Board {
    grid :: Array Coordinate DiscState,
    boardDim :: Int
} deriving (Show)

-- For purposes of Minimax AI, we will need to measure the score of a given state. It will be either a win, indeterminate with score of some kind (from the view of who is requesting it), or a loss
data BoardScore = Win | Indeterminate UnitScore | Lose deriving (Eq, Show)
```

```haskell

-- Define a order for a board score. For least complexity, define ordering as a set of comparative properties between different scores
instance Ord BoardScore where
    (<=) :: BoardScore -> BoardScore -> Bool
    (<=) Lose _ = True -- Lose is the smallest and definitely equal
    (<=) (Indeterminate _) Lose = False -- Indeterminate is never less or equal to a win
    (<=) (Indeterminate _) Win = True -- Indeterminate is always less than a win
    (<=) (Indeterminate a) (Indeterminate b) = a <= b -- For two indeterminates, their respective ordering depends on their scores
    (<=) Win Win = True -- Win is equal with a win
    (<=) Win _ = False -- Otherwise, no

-- Define a scoring function; given a player and a board, what is their score?
score :: Player -> Board -> BoardScore
score player board
    | not (movesPossibleOnBoard board) && redLeading = if player == Red then Win else Lose
    | not (movesPossibleOnBoard board) && blueLeading = if player == Blue then Win else Lose
    | otherwise = Indeterminate (if player == Red then redCount else blueCount)
    where
        redLeading = redCount > blueCount
        blueLeading = blueCount > redCount

        (redCount, blueCount) = pieceCount board

data GameSetup = GameSetup {
    board :: Board, -- Board
    aiPlays :: [Player], -- Which turns AI plays?
    searchDepth :: Int -- Search depth
} deriving (Show)
-- Core functions

-- Count of pieces, per color, for the board
pieceCount :: Board -> (Int, Int)
pieceCount board = foldr adder (0,0) (elems $ grid board) -- Add element by element, start with zero scores for both
    where
        adder :: DiscState -> (Int, Int) -> (Int, Int)
        adder state count@(red, blue) = case state of
            Empty -> count
            Placed Red -> (red+1, blue)
            Placed Blue -> (red, blue+1)

-- Definition of the opposing player for a given player
opposingPlayer :: Player -> Player
opposingPlayer Red = Blue
opposingPlayer Blue = Red

```

```haskell
-- Calculating which player is winning by their score; this does not consider if there are more turns remaining
playerWithBestScore :: Board -> Maybe Player
playerWithBestScore board
    | red == blue = Nothing
    | otherwise = if red > blue then Just Red else Just Blue
    where
        (red, blue) = pieceCount board

-- Define an indexing operation for a board, quite alike what Arrays have
(!) :: Board -> Coordinate -> DiscState
board ! coordinate = (Data.Array.!) (grid board) coordinate

-- Define a validity check operator for indexes; this will return true if the index is acceptable
(!?) :: Board -> Coordinate -> Bool
board !? coord = inRange (bounds $ grid board) coord

-- Define a grid comprehension function; mapping over coordinates of a grid, construct some array of data
comprehensionByBoard :: Board -> (Coordinate -> x) -> [x]
comprehensionByBoard board = comprehensionByDim size
    where
        size = boardDim board

comprehensionByDim :: Int -> (Coordinate -> x) -> [x]
comprehensionByDim size func = [ func (a,b) | a <- [0..size-1], b <- [0..size-1]]

initialGameState :: Int -> [Player] -> Int -> GameSetup
initialGameState dim aiPlays searchDepth = GameSetup { aiPlays, searchDepth, board }
    where
        board = Board { boardDim = dim, grid = array ((0, 0), (dim-1,dim-1)) (comprehensionByDim dim (\p -> (p, startPieces p))) }

        startPieces :: Coordinate -> DiscState
        startPieces (cx, cy)
            | cx == (dim `div` 2) - 1 && cy == (dim `div` 2) - 1 = Placed Blue
            | cx == (dim `div` 2) && cy == (dim `div` 2) = Placed Blue
            | cx == (dim `div` 2) - 1 && cy == (dim `div` 2) = Placed Red
            | cx == (dim `div` 2) && cy == (dim `div` 2) - 1 = Placed Red
            | otherwise = Empty
---- Moves and AI

-- For a given board, player and coordinate, determine what coordinates should be changed to player's color. If an empty list is returned, move is not valid
-- This also includes the starting point given
getMovesOnPoint :: Board -> Player -> Coordinate -> [Coordinate]
getMovesOnPoint board player startCoord@(sx, sy)
    | not (board !? startCoord) = [] -- Coordinate is not a valid position
    | startPiece /= Empty = [] -- Starting piece is not empty
    | null resultSteps = [] -- No valid steps exist
```

```haskell
    | otherwise = startCoord : resultSteps
    where
        resultSteps = concatMap walkAndMark directions -- Valid steps are a concatenation of walked directions - per rules, we can and must mark all branched paths

        walkAndMark :: Coordinate -> [Coordinate] -- If stepping from a given direction, what can we mark (excluding start position)?
        walkAndMark dir@(dx, dy) = walkAndMark' (sx+dx, sy+dy) dir []

        walkAndMark' :: Coordinate -> Coordinate -> [Coordinate] -> [Coordinate] -- Current position, direction, found already
        walkAndMark' cur@(cx, cy) dir@(dx, dy) found
            | isEndPiece = found -- End piece, terminate here
            | isValidTraversalPiece = walkAndMark' (cx+dx, cy+dy) dir (cur:found) -- Can traverse, step forward
            | otherwise = [] -- No valid way to travel nor end, return nothing
            where
                isEndPiece = isValidPos && board ! cur == Placed player
                isValidTraversalPiece = isValidPos && board ! cur == Placed (opposingPlayer player)
                isValidPos = board !? cur

        directions = [(-1,-1), (-1,0), (-1,1), (0,-1), (0,1), (1,-1), (1,0), (1,1)] -- Which directions need to be checked. As defined in the rules, we work on either horizontal, vertical or diagonal lines
        startPiece = board ! startCoord

movesForPlayer :: Board -> Player -> [[Coordinate]]  -- Return all movesets that are nonempty, for a given player?
movesForPlayer board player = filter (not . null) mappedCoords
    where
        mappedCoords = comprehensionByBoard board (getMovesOnPoint board player)

movesPossibleOnBoard :: Board -> Bool -- Are there any moves possible on board?
movesPossibleOnBoard board = not (null (movesForPlayer board Red) && null (movesForPlayer board Blue))

applyMove :: Board -> Player -> [Coordinate] -> Board
applyMove board player moveList = board { grid = grid board // map (, Placed player) moveList }

-- AI

getAIMove :: GameSetup -> Player -> Int -> [Coordinate]
getAIMove setup player depth
    | null possibleMoves = [] -- No moves available
    | otherwise = fst $ maximumBy (comparing snd) evaluatedMoves
  where
    -- Get all possible moves for the current player
    possibleMoves = movesForPlayer (board setup) player

```

```haskell
    -- Evaluate each move using minimax
    evaluatedMoves = [(move, minimax (applyMove (board setup) player move) (opposingPlayer player) (depth - 1)) | move <- possibleMoves]

-- Minimax algorithm for game tree evaluation
minimax :: Board -> Player -> Int -> Int
minimax board player depth
    | depth == 0 || null possibleMoves = evaluateBoard board player -- Base case: evaluate board
    | player == maximizingPlayer = maximum [minimax (applyMove board player move) (opposingPlayer player) (depth - 1) | move <- possibleMoves]
    | otherwise = minimum [minimax (applyMove board player move) (opposingPlayer player) (depth - 1) | move <- possibleMoves]
  where
    possibleMoves = movesForPlayer board player
    maximizingPlayer = Red -- Assume Red is the maximizing player

-- Example board evaluation function
evaluateBoard :: Board -> Player -> Int
evaluateBoard board player = scoreForPlayer - scoreForOpponent
  where
    scoreForPlayer = length [pos | (pos, state) <- assocs (grid board), state == Placed player]
    scoreForOpponent = length [pos | (pos, state) <- assocs (grid board), state == Placed (opposingPlayer player)]

```