

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.

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

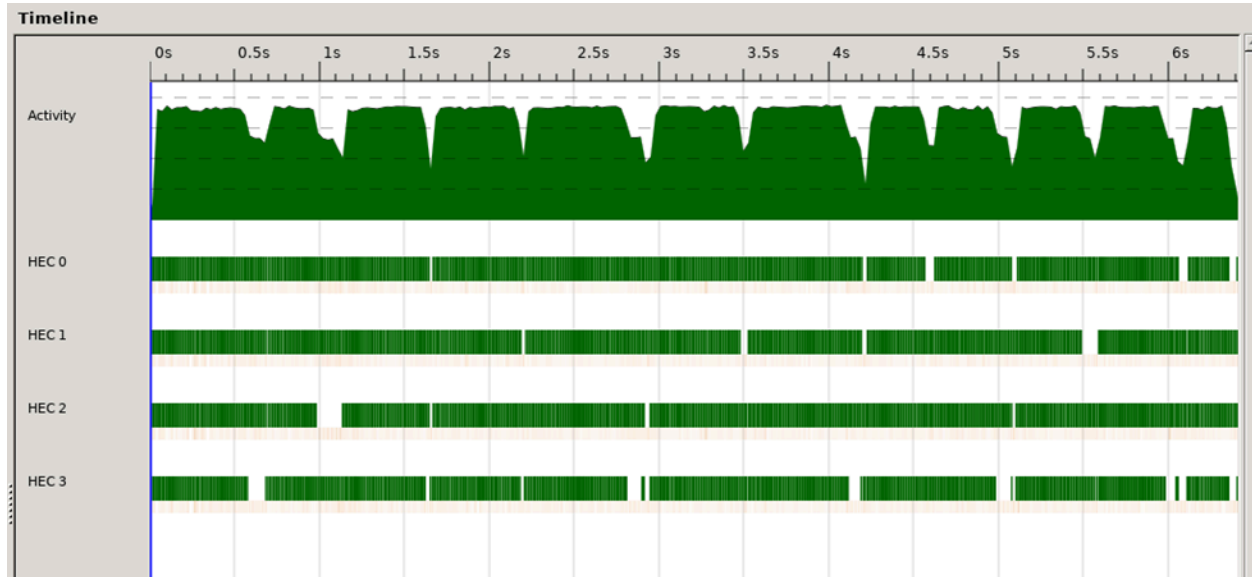
$$\frac{P}{N} - P = \frac{1}{S} - 1$$

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

E.g., for parMap, when N=4, $S = .798(\text{runtime } N=1) / .637(\text{runtime } N=4) = 1.252$, $P = 0.267$

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 pre-made 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).

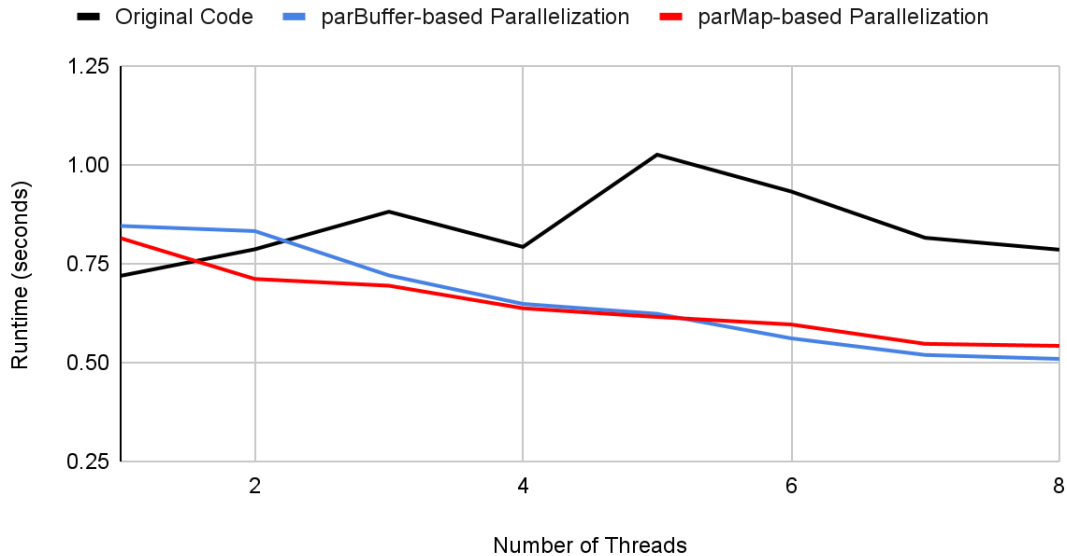


Threadscope graph with depth = 6, parallelDepth =5, threads = 4

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:

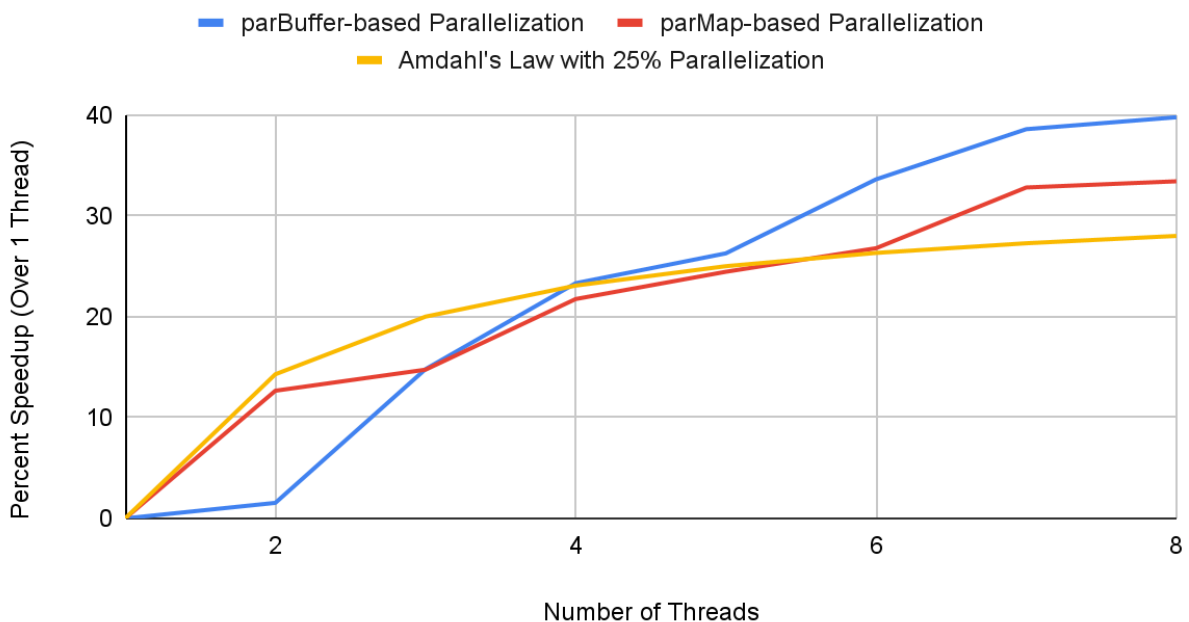
Performance of Decision-Making Process vs Threads Given



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

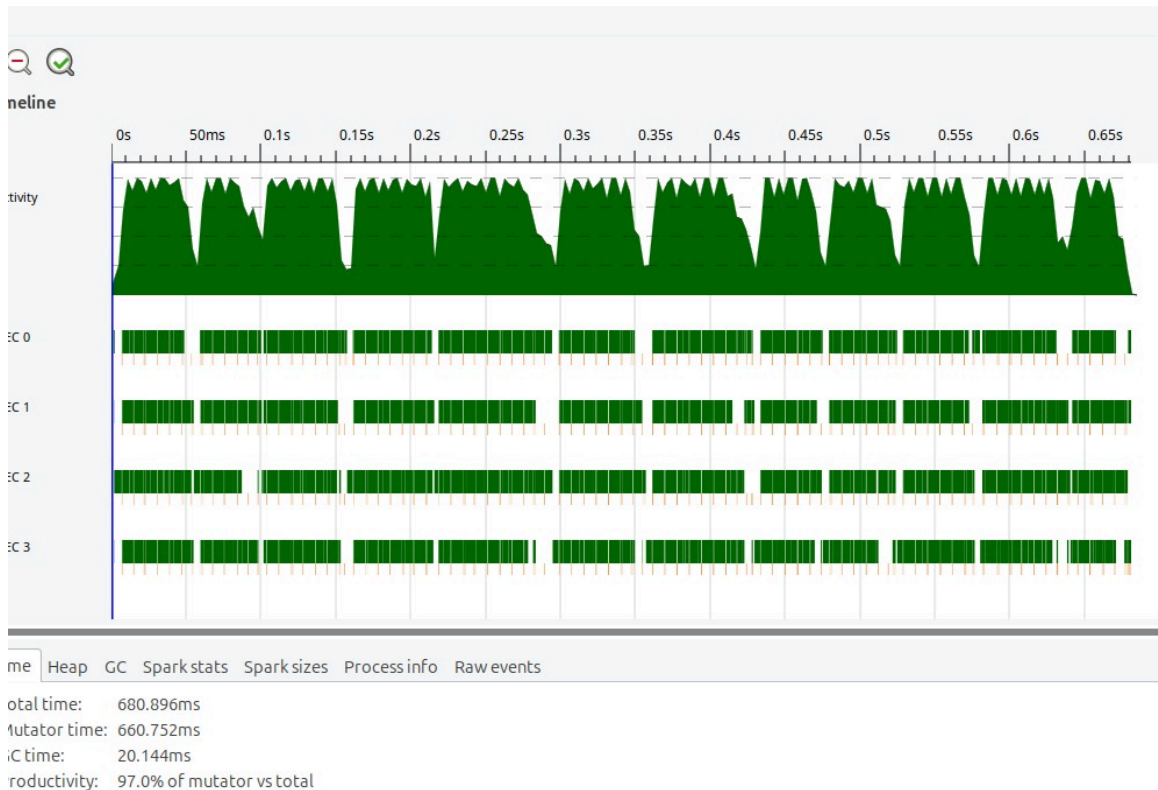
While these performance gains over the original code are significant, the intention behind both the parMap and parBuffer approaches is clear when you compare their performance at each thread count against its single-threaded execution time: When given at least 3 threads, both of these approaches are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

Decision-Making Process Performance Gains

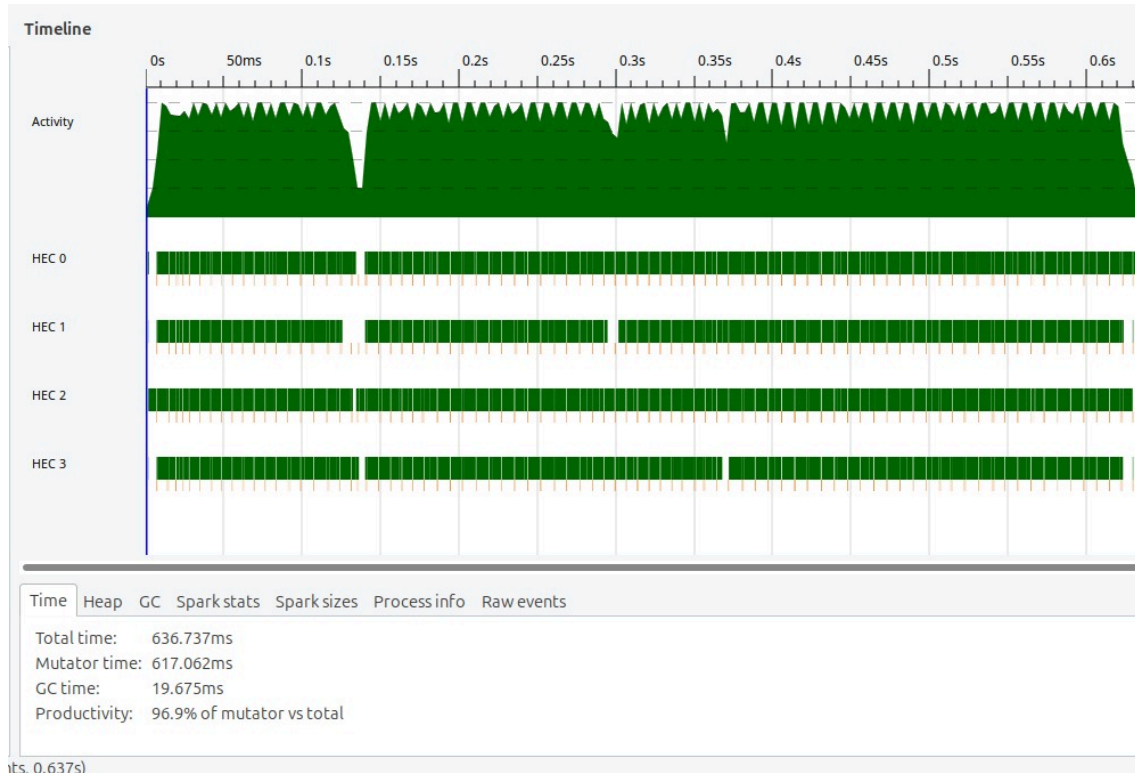


All data collected while running at depth = 5, parallelDepth = 4, starting board = custom_game_2

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

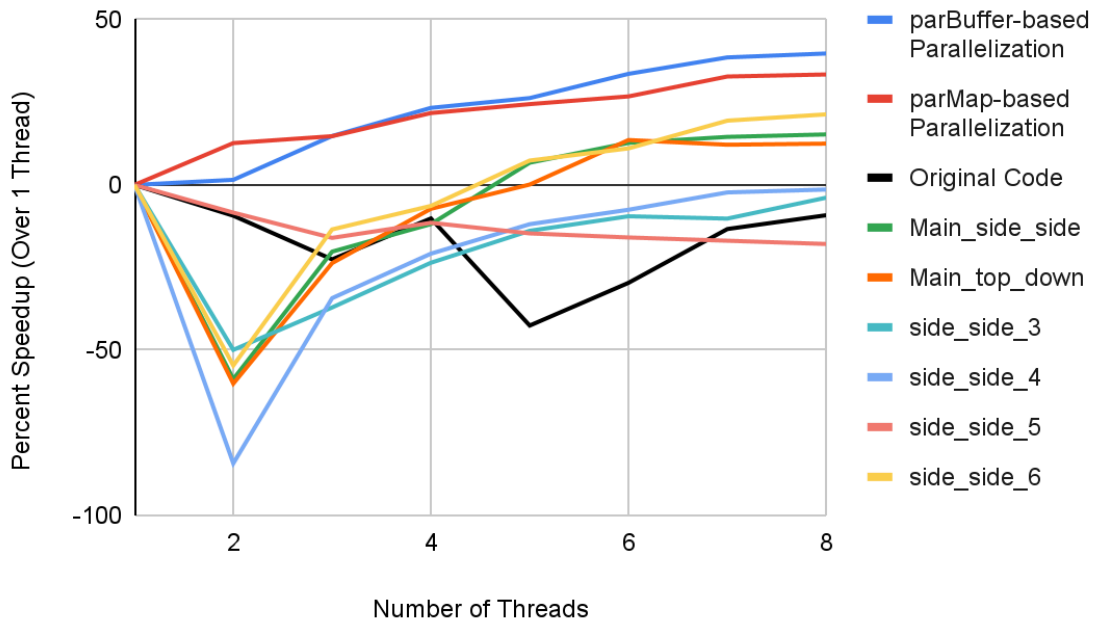


Threadscope graph for the parBuffer-based parallelization with 4 threads

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.

Parallelization Comparison for all Approaches



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 line 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 parallelization 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
├── hackage-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

```

1  {-# LANGUAGE BangPatterns #-}
2
3  import System.Environment (getArgs)
4  import System.IO (readFile)
5  import Data.Array (array, listArray, assocs)
6  import Data.Foldable (maximumBy)
7  import Data.Ord (comparing)
8  import Control.Parallel.Strategies (parMap, rpar, rseq, using, parList, rdeepseq)
9  import Othello.GameLogic (
10     GameSetup(..), Player(..), initialGameState, Board(..), DiscState(..),
11     movesForPlayer, applyMove, Coordinate, opposingPlayer
12 )
13
14 -- Parse a custom board state from a string
15 parseCustomBoard :: String -> Int -> Board
16 parseCustomBoard input dim = Board {
17     grid = listArray ((0, 0), (dim-1, dim-1)) discStates,
18     boardDim = dim
19 }
20 where
21     rows = lines input
22     discStates = concatMap parseRow rows
23     parseRow row = map parseDiscState (words row)
24     parseDiscState "E" = Empty
25     parseDiscState "R" = Placed Red

```

```

26     parseDiscState "B" = Placed Blue
27     parseDiscState _ = error "Invalid disc state in custom board. Use 'E', 'R', or
'B'."
28
29 -- Define a simple entry point for the program
30 main :: IO ()
31 main = do
32     -- Get command-line arguments
33     args <- getArgs
34
35     -- Parse depth arguments
36     let (depth, parallelDepth, inputFile) = case args of
37         (d:p:file:_) -> (read d, read p, Just file)
38         (d:p:_)      -> (read d, read p, Nothing)
39         _             -> error "Usage: <depth> <parallelDepth> [gameboard file]"
40
41     -- Load the custom board or use the default initial game state
42     gameSetup <- case inputFile of
43         Just file -> do
44             content <- readFile file
45             let customBoard = parseCustomBoard content 8 -- Assume an 8x8 board for
simplicity
46                 return GameSetup {
47                     board = customBoard,
48                     aiPlays = [Red, Blue],
49                     searchDepth = depth
50                 }
51         Nothing -> return (initialGameState 8 [Red, Blue] depth)
52
53     putStrLn $ "Using minimax depth: " ++ show depth
54     putStrLn $ "Parallelizing at depth: " ++ show parallelDepth
55
56     let currentPlayer = Red
57
58     let moves = getAIMove gameSetup currentPlayer depth parallelDepth
59     if null moves
60         then putStrLn "No valid moves available."
61         else do
62             let selectedMove = head moves
63                 let updatedBoard = applyMove (board gameSetup) currentPlayer [selectedMove]
64                     putStrLn $ "Next move: " ++ show selectedMove
65                     putStrLn "Updated board state:"
66                     print updatedBoard
67
68 -- Define a custom AI move function using alpha-beta pruning
69 getAIMove :: GameSetup -> Player -> Int -> Int -> [Coordinate]
70 getAIMove setup player depth parallelDepth
71     | null possibleMoves = [] -- No moves available
72     | otherwise = fst $ maximumBy (comparing snd) evaluatedMoves
73 where

```

```

74 possibleMoves = movesForPlayer (board setup) player
75
76 evaluatedMoves =
77     if depth >= parallelDepth
78     then parallelEvaluate possibleMoves
79     else sequentialEvaluate possibleMoves
80
81 sequentialEvaluate moves =
82     [ (move, minimax (applyMove (board setup) player move) (opposingPlayer player)
(depth - 1) minBound maxBound)
83     | move <- moves
84     ]
85
86 parallelEvaluate moves =
87     let groupedMoves = groupByThread possibleMoves (length moves)
88         evalGroup group =
89             [ (move, minimax (applyMove (board setup) player move) (opposingPlayer
player) (depth - 1) minBound maxBound)
90             | (move, _) <- group
91             ]
92     in concat $ parMap rdeepseq evalGroup groupedMoves
93
94 groupByThread moves numThreads =
95     [ [ (move, idx) | (move, idx) <- zip moves [0..], idx `mod` numThreads ==
threadIdx ]
96     | threadIdx <- [0..numThreads - 1]
97     ]
98
99
100 -- Minimax algorithm with alpha-beta pruning
101 minimax :: Board -> Player -> Int -> Int -> Int -> Int
102 minimax board player depth alpha beta
103     | depth == 0 || null possibleMoves = evaluateBoard board player --base case
104     | player == maximizingPlayer = maximize alpha beta possibleMoves
105     | otherwise = minimize alpha beta possibleMoves
106 where
107     possibleMoves = movesForPlayer board player
108     maximizingPlayer = Red
109
110     maximize :: Int -> Int -> [[Coordinate]] -> Int
111     maximize a b [] = a
112     maximize a b (move:moves)
113         | a' >= b = a'
114         | otherwise = maximize a' b moves
115     where
116     a' = max a (minimax (applyMove board player move) (opposingPlayer player)
(depth - 1) a b)
117
118     minimize :: Int -> Int -> [[Coordinate]] -> Int
119     minimize a b [] = b
120     minimize a b (move:moves)

```

```

121     | a >= b' = b'
122     | otherwise = minimize a b' moves
123     where
124         b' = min b (minimax (applyMove board player move) (opposingPlayer player)
(depth - 1) a b)
125
126 -- Board evaluation
127 evaluateBoard :: Board -> Player -> Int
128 evaluateBoard board player = scoreForPlayer - scoreForOpponent
129     where
130         scoreForPlayer = length [pos | (pos, state) <- assocs (grid board), state == Placed
player]
131         scoreForOpponent = length [pos | (pos, state) <- assocs (grid board), state ==
Placed (opposingPlayer player)]
132
133
134

```

othello/app/Main_top_down.hs

```

1 {-# LANGUAGE BangPatterns #-}
2
3 import System.Environment (getArgs)
4 import System.IO (readFile)
5 import Data.Array (array, listArray, assocs)
6 import Data.Foldable (maximumBy)
7 import Data.Ord (comparing)
8 import Control.Parallel.Strategies (parMap, rpar, rseq, using, parList, rdeepseq)
9 import Othello.GameLogic (
10     GameSetup(..), Player(..), initialState, Board(..), DiscState(..),
11     movesForPlayer, applyMove, Coordinate, opposingPlayer
12 )
13
14 -- Parse a custom board state from a string
15 parseCustomBoard :: String -> Int -> Board
16 parseCustomBoard input dim = Board {
17     grid = listArray ((0, 0), (dim-1, dim-1)) discStates,
18     boardDim = dim
19 }
20 where
21     rows = lines input
22     discStates = concatMap parseRow rows
23     parseRow row = map parseDiscState (words row)
24     parseDiscState "E" = Empty
25     parseDiscState "R" = Placed Red
26     parseDiscState "B" = Placed Blue
27     parseDiscState _ = error "Invalid disc state in custom board. Use 'E', 'R', or
'B' ."
28
29 -- Define a simple entry point for the program
30 main :: IO ()

```

```

31 main = do
32   -- Get command-line arguments
33   args <- getArgs
34
35   -- Parse depth arguments
36   let (depth, parallelDepth, inputFile) = case args of
37     (d:p:file:_) -> (read d, read p, Just file)
38     (d:p:_)      -> (read d, read p, Nothing)
39     _            -> error "Usage: <depth> <parallelDepth> [gameboard file]"
40
41   -- Load the custom board or use the default initial game state
42   gameSetup <- case inputFile of
43     Just file -> do
44       content <- readFile file
45       let customBoard = parseCustomBoard content 8 -- Assume an 8x8 board for
simplicity
46         return GameSetup {
47           board = customBoard,
48           aiPlays = [Red, Blue],
49           searchDepth = depth
50         }
51     Nothing -> return (initialGameState 8 [Red, Blue] depth)
52
53   putStrLn $ "Using minimax depth: " ++ show depth
54   putStrLn $ "Parallelizing at depth: " ++ show parallelDepth
55
56   let currentPlayer = Red
57   let moves = getAIMove gameSetup currentPlayer depth parallelDepth
58   if null moves
59     then putStrLn "No valid moves available."
60     else do
61       let selectedMove = head moves
62       let updatedBoard = applyMove (board gameSetup) currentPlayer [selectedMove]
63       putStrLn $ "Next move: " ++ show selectedMove
64       putStrLn "Updated board state:"
65       print updatedBoard
66
67   -- Define a custom AI move function using alpha-beta pruning
68   getAIMove :: GameSetup -> Player -> Int -> Int -> [Coordinate]
69   getAIMove setup player depth parallelDepth
70     | null possibleMoves = [] -- No moves available
71     | otherwise = fst $ maximumBy (comparing snd) evaluatedMoves
72   where
73     possibleMoves = movesForPlayer (board setup) player
74
75     evaluatedMoves =
76       if depth >= parallelDepth
77         then parallelEvaluate possibleMoves
78         else sequentialEvaluate possibleMoves
79

```

```

80     sequentialEvaluate moves =
81         [(move, minimax (applyMove (board setup) player move) (opposingPlayer player)
(depth - 1) minBound maxBound)
82         | move <- moves]
83
84     parallelEvaluate moves =
85         let eval move =
86             (move, minimax (applyMove (board setup) player move) (opposingPlayer
player) (depth - 1) minBound maxBound)
87         in parMap rdeepseq eval moves
88
89 -- Minimax algorithm with alpha-beta pruning
90 minimax :: Board -> Player -> Int -> Int -> Int -> Int
91 minimax board player depth alpha beta
92     | depth == 0 || null possibleMoves = evaluateBoard board player --Base case
93     | player == maximizingPlayer = maximize alpha beta possibleMoves
94     | otherwise = minimize alpha beta possibleMoves
95 where
96     possibleMoves = movesForPlayer board player
97     maximizingPlayer = Red
98
99     maximize :: Int -> Int -> [[Coordinate]] -> Int
100    maximize a b [] = a
101    maximize a b (move:moves)
102        | a' >= b = a'
103        | otherwise = maximize a' b moves
104    where
105        a' = max a (minimax (applyMove board player move) (opposingPlayer player)
(depth - 1) a b)
106
107    minimize :: Int -> Int -> [[Coordinate]] -> Int
108    minimize a b [] = b
109    minimize a b (move:moves)
110        | a >= b' = b'
111        | otherwise = minimize a b' moves
112    where
113        b' = min b (minimax (applyMove board player move) (opposingPlayer player)
(depth - 1) a b)
114
115 -- Board evaluation
116 evaluateBoard :: Board -> Player -> Int
117 evaluateBoard board player = scoreForPlayer - scoreForOpponent
118 where
119     scoreForPlayer = length [pos | (pos, state) <- assocs (grid board), state == Placed
player]
120     scoreForOpponent = length [pos | (pos, state) <- assocs (grid board), state ==
Placed (opposingPlayer player)]
121
122
123

```

othello/app/side_side_3.hs

```
1 {-# LANGUAGE BangPatterns #-}
2
3 import System.Environment (getArgs)
4 import System.IO (readFile)
5 import Data.Array (array, listArray, assocs)
6 import Data.Foldable (maximumBy)
7 import Data.Ord (comparing)
8 import Control.Parallel.Strategies (parMap, parListChunk, rdeepseq)
9 import Othello.GameLogic (
10     GameSetup(..), Player(..), initialGameState, Board(..), DiscState(..),
11     movesForPlayer, applyMove, Coordinate, opposingPlayer
12 )
13 import GHC.Conc (numCapabilities)
14 import Data.List.Split (chunksOf)
15
16 -- Parse a custom board state from a string
17 parseCustomBoard :: String -> Int -> Board
18 parseCustomBoard input dim = Board {
19     grid = listArray ((0, 0), (dim-1, dim-1)) discStates,
20     boardDim = dim
21 }
22 where
23     rows = lines input
24     discStates = concatMap parseRow rows
25     parseRow row = map parseDiscState (words row)
26     parseDiscState "E" = Empty
27     parseDiscState "R" = Placed Red
28     parseDiscState "B" = Placed Blue
29     parseDiscState _ = error "Invalid disc state in custom board. Use 'E', 'R', or
30 'B'."
31
32 -- Define a simple entry point for the program
33 main :: IO ()
34 main = do
35     -- Get command-line arguments
36     args <- getArgs
37
38     -- Parse depth arguments
39     let (depth, parallelDepth, inputFile) = case args of
40         (d:p:file:_) -> (read d, read p, Just file)
41         (d:p:_)      -> (read d, read p, Nothing)
42         _            -> error "Usage: <depth> <parallelDepth> [gameboard file]"
43
44     -- Load the custom board or use the default initial game state
45     gameSetup <- case inputFile of
46         Just file -> do
47             content <- readFile file
```



```

48         let customBoard = parseCustomBoard content 8 -- Assume an 8x8 board for
simplicity
49         return GameSetup {
50             board = customBoard,
51             aiPlays = [Red, Blue],
52             searchDepth = depth
53         }
54         Nothing -> return (initialGameState 8 [Red, Blue] depth)
55
56     putStrLn $ "Using minimax depth: " ++ show depth
57     putStrLn $ "Parallelizing at depth: " ++ show parallelDepth
58
59     let currentPlayer = Red
60
61     let moves = getAIMove gameSetup currentPlayer depth parallelDepth
62     if null moves
63     then putStrLn "No valid moves available."
64     else do
65         let selectedMove = head moves
66             updatedBoard = applyMove (board gameSetup) currentPlayer [selectedMove]
67         putStrLn $ "Next move: " ++ show selectedMove
68         putStrLn "Updated board state:"
69         print updatedBoard
70
71 -- Define a custom AI move function using alpha-beta pruning
72 getAIMove :: GameSetup -> Player -> Int -> Int -> [Coordinate]
73 getAIMove setup player depth parallelDepth
74     | null possibleMoves = [] -- No moves available
75     | otherwise = fst $ maximumBy (comparing snd) evaluatedMoves
76 where
77     possibleMoves = movesForPlayer (board setup) player
78
79     evaluatedMoves =
80         if depth == parallelDepth
81         then parallelEvaluate possibleMoves
82         else sequentialEvaluate possibleMoves
83
84     sequentialEvaluate moves =
85         [(move, minimax (applyMove (board setup) player move) (opposingPlayer player)
(depth - 1) parallelDepth minBound maxBound)
86         | move <- moves]
87
88     parallelEvaluate moves =
89         let chunkSize = max 1 (length moves `div` numCapabilities) -- Split moves into
chunks
90             eval move = (move, minimax (applyMove (board setup) player move)
(opposingPlayer player) (depth - 1) parallelDepth minBound maxBound)
91         in concat $ parMap rdeepseq (map eval) (chunksOf chunkSize moves)
92
93 -- Minimax algorithm with alpha-beta pruning
94 minimax :: Board -> Player -> Int -> Int -> Int -> Int -> Int

```

```

95 minimax board player depth parallelDepth alpha beta
96   | depth == 0 || null possibleMoves = evaluateBoard board player
97   | player == maximizingPlayer = maximize alpha beta possibleMoves
98   | otherwise = minimize alpha beta possibleMoves
99 where
100   possibleMoves = movesForPlayer board player
101   maximizingPlayer = Red
102
103   maximize :: Int -> Int -> [[Coordinate]] -> Int
104   maximize a b [] = a
105   maximize a b (move:moves)
106     | a' >= b = a'
107     | otherwise = maximize a' b moves
108   where
109     a' = max a (nextEval move a b)
110
111   minimize :: Int -> Int -> [[Coordinate]] -> Int
112   minimize a b [] = b
113   minimize a b (move:moves)
114     | a >= b' = b'
115     | otherwise = minimize a b' moves
116   where
117     b' = min b (nextEval move a b)
118
119   nextEval move a b
120     | depth == parallelDepth =
121       let results = parMap rdeepseq eval possibleMoves
122       in if player == maximizingPlayer then maximum results else minimum results
123     | otherwise =
124       minimax (applyMove board player move) (opposingPlayer player) (depth - 1)
parallelDepth a b
125   where
126     eval move = minimax (applyMove board player move) (opposingPlayer player)
(depth - 1) parallelDepth a b
127
128
129
130 -- Board evaluation
131 evaluateBoard :: Board -> Player -> Int
132 evaluateBoard board player = scoreForPlayer - scoreForOpponent
133   where
134     scoreForPlayer = length [pos | (pos, state) <- assocs (grid board), state == Placed
player]
135     scoreForOpponent = length [pos | (pos, state) <- assocs (grid board), state ==
Placed (opposingPlayer player)]
136
137
138

```

othello/app/side_side_4.hs

```

1 {-# LANGUAGE BangPatterns #-}
2
3 import System.Environment (getArgs)
4 import System.IO (readFile)
5 import Data.Array (array, listArray, assocs)
6 import Data.Foldable (maximumBy)
7 import Data.Ord (comparing)
8 import Control.Parallel.Strategies (parMap, parListChunk, rdeepseq)
9 import Othello.GameLogic (
10     GameSetup(..), Player(..), initialGameState, Board(..), DiscState(..),
11     movesForPlayer, applyMove, Coordinate, opposingPlayer
12 )
13 import GHC.Conc (numCapabilities)
14 import Data.List.Split (chunksOf)
15
16 -- Parse a custom board state from a string
17 parseCustomBoard :: String -> Int -> Board
18 parseCustomBoard input dim = Board {
19     grid = listArray ((0, 0), (dim-1, dim-1)) discStates,
20     boardDim = dim
21 }
22 where
23     rows = lines input
24     discStates = concatMap parseRow rows
25     parseRow row = map parseDiscState (words row)
26     parseDiscState "E" = Empty
27     parseDiscState "R" = Placed Red
28     parseDiscState "B" = Placed Blue
29     parseDiscState _ = error "Invalid disc state in custom board. Use 'E', 'R', or
30 'B'."
31
32 -- Main entry point
33 -- Define a simple entry point for the program
34 main :: IO ()
35 main = do
36     -- Get command-line arguments
37     args <- getArgs
38
39     -- Parse depth arguments
40     let (depth, parallelDepth, inputFile) = case args of
41         (d:p:file:_) -> (read d, read p, Just file)
42         (d:p:_)      -> (read d, read p, Nothing)
43         _            -> error "Usage: <depth> <parallelDepth> [gameboard file]"
44
45     -- Load the custom board or use the default initial game state
46     gameSetup <- case inputFile of
47         Just file -> do
48             content <- readFile file
49             let customBoard = parseCustomBoard content 8 -- Assume an 8x8 board for
simplicity

```

```

49     return GameSetup {
50         board = customBoard,
51         aiPlays = [Red, Blue],
52         searchDepth = depth
53     }
54     Nothing -> return (initialGameState 8 [Red, Blue] depth)
55
56     putStrLn $ "Using minimax depth: " ++ show depth
57     putStrLn $ "Parallelizing at depth: " ++ show parallelDepth
58
59     let currentPlayer = Red
60
61     let moves = getAIMove gameSetup currentPlayer depth parallelDepth
62     if null moves
63     then putStrLn "No valid moves available."
64     else do
65         let selectedMove = head moves
66             updatedBoard = applyMove (board gameSetup) currentPlayer [selectedMove]
67         putStrLn $ "Next move: " ++ show selectedMove
68         putStrLn "Updated board state:"
69         print updatedBoard
70
71
72 getAIMove :: GameSetup -> Player -> Int -> Int -> [Coordinate]
73 getAIMove setup player depth parallelDepth
74     | null possibleMoves = [] -- No moves available
75     | otherwise = fst $ maximumBy (comparing snd) evaluatedMoves
76 where
77     possibleMoves = movesForPlayer (board setup) player
78
79     evaluatedMoves =
80         if depth == parallelDepth
81         then parallelEvaluate possibleMoves
82         else sequentialEvaluate possibleMoves
83
84     sequentialEvaluate moves =
85         [(move, minimax (applyMove (board setup) player move) (opposingPlayer player)
86 (depth - 1) parallelDepth minBound maxBound)
87         | move <- moves]
88
89     parallelEvaluate moves =
90         let chunkSize = max 1 (length moves `div` numCapabilities) -- Split moves into
91 chunks
92             eval move = (move, minimax (applyMove (board setup) player move)
93 (opposingPlayer player) (depth - 1) parallelDepth minBound maxBound)
94         in concat $ parMap rdeepseq (map eval) (chunksOf chunkSize moves)
95
96 -- Minimax algorithm with alpha-beta pruning
97 minimax :: Board -> Player -> Int -> Int -> Int -> Int -> Int
98 minimax board player depth parallelDepth alpha beta
99     | depth == 0 || null possibleMoves = evaluateBoard board player --Base case

```

```

97 | | player == maximizingPlayer = maximize alpha beta possibleMoves
98 | | otherwise = minimize alpha beta possibleMoves
99 | where
100 | possibleMoves = movesForPlayer board player
101 | maximizingPlayer = Red
102 |
103 | maximize :: Int -> Int -> [[Coordinate]] -> Int
104 | maximize a b [] = a
105 | maximize a b (move:moves)
106 |   | a' >= b = a'
107 |   | otherwise = maximize a' b moves
108 |   where
109 |     a' = max a (nextEval move a b)
110 |
111 | minimize :: Int -> Int -> [[Coordinate]] -> Int
112 | minimize a b [] = b
113 | minimize a b (move:moves)
114 |   | a >= b' = b'
115 |   | otherwise = minimize a b' moves
116 |   where
117 |     b' = min b (nextEval move a b)
118 |
119 | nextEval move a b
120 |   | depth == parallelDepth =
121 |     let results = parMap rdeepseq eval possibleMoves
122 |     in if player == maximizingPlayer then maximum results else minimum results
123 |   | otherwise =
124 |     minimax (applyMove board player move) (opposingPlayer player) (depth - 1)
parallelDepth a b
125 |     where
126 |       eval move = minimax (applyMove board player move) (opposingPlayer player)
(depth - 1) parallelDepth a b
127 |
128 |
129 |
130 | -- Board evaluation
131 | evaluateBoard :: Board -> Player -> Int
132 | evaluateBoard board player = scoreForPlayer - scoreForOpponent
133 |   where
134 |     scoreForPlayer = length [pos | (pos, state) <- assocs (grid board), state == Placed
player]
135 |     scoreForOpponent = length [pos | (pos, state) <- assocs (grid board), state ==
Placed (opposingPlayer player)]
136 |
137 |

```

othello/app/side_side_5.hs

```

1 | module Main where
2 |
3 | import GHC.Conc (numCapabilities)

```

```

4 import System.Environment (getArgs)
5 import System.IO (readFile)
6 import Data.Array (array, listArray, assocs)
7 import Data.Foldable (maximumBy)
8 import Data.Ord (comparing)
9 import Control.Parallel (par, pseq)
10 import Control.Parallel.Strategies (parList, parListChunk, rdeepseq, using, parMap)
11 import Othello.GameLogic (
12     GameSetup(..), Player(..), initialGameState, Board(..), DiscState(..),
13     movesForPlayer, applyMove, Coordinate, opposingPlayer
14 )
15
16 -- Parse a custom board state from a string
17 parseCustomBoard :: String -> Int -> Board
18 parseCustomBoard input dim = Board {
19     grid = listArray ((0, 0), (dim-1, dim-1)) discStates,
20     boardDim = dim
21 }
22 where
23     rows = lines input
24     discStates = concatMap parseRow rows
25     parseRow row = map parseDiscState (words row)
26     parseDiscState "E" = Empty
27     parseDiscState "R" = Placed Red
28     parseDiscState "B" = Placed Blue
29     parseDiscState _ = error "Invalid disc state in custom board. Use 'E', 'R', or
30 'B'."
31
32 -- Define a simple entry point for the program
33 main :: IO ()
34 main = do
35     -- Get command-line arguments
36     args <- getArgs
37
38     -- Parse depth arguments
39     let (depth, parallelDepth, inputFile) = case args of
40         (d:p:file:_) -> (read d, read p, Just file)
41         (d:p:_)      -> (read d, read p, Nothing)
42         _            -> error "Usage: <depth> <parallelDepth> [gameboard file]"
43
44     -- Load the custom board or use the default initial game state
45     gameSetup <- case inputFile of
46         Just file -> do
47             content <- readFile file
48             let customBoard = parseCustomBoard content 8 -- Assume an 8x8 board for
49                 simplicity
50             return GameSetup {
51                 board = customBoard,
52                 aiPlays = [Red, Blue],
53                 searchDepth = depth

```

```

52     }
53     Nothing -> return (initialGameState 8 [Red, Blue] depth)
54
55     putStrLn $ "Using minimax depth: " ++ show depth
56     putStrLn $ "Parallelizing at depth: " ++ show parallelDepth
57
58     let currentPlayer = Red
59
60     -- Calculate the next move for the current player with the specified depth
61     let moves = getAIMove gameSetup currentPlayer depth parallelDepth
62     if null moves
63     then putStrLn "No valid moves available."
64     else do
65         let selectedMove = head moves
66         let updatedBoard = applyMove (board gameSetup) currentPlayer [selectedMove]
67         putStrLn $ "Next move: " ++ show selectedMove
68         putStrLn "Updated board state:"
69         print updatedBoard
70
71     -- Minimax with optional parallelization
72     minimax :: Board -> Player -> Int -> Int -> Int -> Int -> Int
73     minimax board player depth alpha beta parallelDepth
74         | depth == 0 || null possibleMoves = evaluateBoard board player -- Base case
75         | depth == parallelDepth = parallelMinimaxAggressive board player depth alpha beta
76         | otherwise = sequentialMinimax possibleMoves alpha beta
77     where
78         possibleMoves = movesForPlayer board player
79         maximizingPlayer = Red -- Assume Red is the maximizing player
80
81     -- Sequential minimax
82     sequentialMinimax :: [[Coordinate]] -> Int -> Int -> Int
83     sequentialMinimax [] a _ = a -- No more moves, return alpha
84     sequentialMinimax (move:moves) a b
85         | a >= b = a -- Prune the rest of the tree
86         | otherwise = sequentialMinimax moves a' b
87     where
88         a' = max a (evaluateMove board player (depth - 1) a b move)
89
90     evaluateMove :: Board -> Player -> Int -> Int -> Int -> [Coordinate] -> Int
91     evaluateMove b p d a b' move = minimax (applyMove b p move) (opposingPlayer p) d a
92     b' parallelDepth
93
94     -- parallelMinimax
95     parallelMinimaxAggressive :: Board -> Player -> Int -> Int -> Int -> [[Coordinate]] ->
96     Int
97     parallelMinimaxAggressive board player depth a b moves =
98         let chunkedMoves = map (:[]) moves `using` parListChunk (max 1 (length moves `div`
99         (round( 1 * fromIntegral numCapabilities)))) rdeepseq
100         results = map (maximum . map (evaluateMove board player (depth - 1) a b))
101         chunkedMoves

```

```

98     in maximum results
99     where
100     evaluateMove :: Board -> Player -> Int -> Int -> Int -> [Coordinate] -> Int
101     evaluateMove b p d a b' move = minimax (applyMove b p move) (opposingPlayer p) d a
b' depth
102
103 -- AI
104 getAIMove :: GameSetup -> Player -> Int -> Int -> [Coordinate]
105 getAIMove setup player depth parallelDepth
106     | null possibleMoves = [] -- No moves available
107     | otherwise = fst $ maximumBy (comparing snd) evaluatedMoves
108     where
109     -- Get all possible moves for the current player
110     possibleMoves = movesForPlayer (board setup) player
111
112     -- Evaluate each move using minimax
113     evaluatedMoves = [
114         (move, minimax (applyMove (board setup) player move) (opposingPlayer player)
(depth - 1) minBound maxBound parallelDepth)
115         | move <- possibleMoves
116     ]
117
118 -- Evaluate Board
119 evaluateBoard :: Board -> Player -> Int
120 evaluateBoard board player = scoreForPlayer - scoreForOpponent
121     where
122     scoreForPlayer = length [pos | (pos, state) <- assocs (grid board), state == Placed
player]
123     scoreForOpponent = length [pos | (pos, state) <- assocs (grid board), state ==
Placed (opposingPlayer player)]
124

```

othello/app/side_side_6.hs

```

1  module Main where
2
3  import System.Environment (getArgs)
4  import System.IO (readFile)
5  import Data.Array (array, listArray, assocs)
6  import Control.Parallel.Strategies (parMap, rdeepseq, parBuffer, using)
7  import Data.List (maximumBy)
8  import Data.Ord (comparing)
9  import Othello.GameLogic (
10     GameSetup(..), Player(..), initialGameState, Board(..), DiscState(..),
11     movesForPlayer, applyMove, Coordinate, opposingPlayer
12     )
13
14 -- Parse a custom board state from a string
15 parseCustomBoard :: String -> Int -> Board
16 parseCustomBoard input dim = Board {
17     grid = listArray ((0, 0), (dim-1, dim-1)) discStates,

```



```

18     boardDim = dim
19 }
20 where
21     rows = lines input
22     discStates = concatMap parseRow rows
23     parseRow row = map parseDiscState (words row)
24     parseDiscState "E" = Empty
25     parseDiscState "R" = Placed Red
26     parseDiscState "B" = Placed Blue
27     parseDiscState _ = error "Invalid disc state in custom board. Use 'E', 'R', or
'B'."
28
29 -- Define a simple entry point for the program
30 main :: IO ()
31 main = do
32     -- Get command-line arguments
33     args <- getArgs
34
35     -- Parse depth arguments
36     let (depth, parallelDepth, inputFile) = case args of
37         (d:p:file:_) -> (read d, read p, Just file)
38         (d:p:_)      -> (read d, read p, Nothing)
39         _            -> error "Usage: <depth> <parallelDepth> [gameboard file]"
40
41     -- Load the custom board or use the default initial game state
42     gameSetup <- case inputFile of
43         Just file -> do
44             content <- readFile file
45             let customBoard = parseCustomBoard content 8 -- Assume an 8x8 board for
simplicity
46                 return GameSetup {
47                     board = customBoard,
48                     aiPlays = [Red, Blue],
49                     searchDepth = depth
50                 }
51         Nothing -> return (initialGameState 8 [Red, Blue] depth)
52
53     putStrLn $ "Using minimax depth: " ++ show depth
54     putStrLn $ "Parallelizing at depth: " ++ show parallelDepth
55
56     let currentPlayer = Red
57
58     -- Calculate the next move for the current player with the specified depth
59     let moves = getAIMoveParallel gameSetup currentPlayer depth parallelDepth
60     if null moves
61     then putStrLn "No valid moves available."
62     else do
63         let selectedMove = head moves
64             updatedBoard = applyMove (board gameSetup) currentPlayer [selectedMove]
65         putStrLn $ "Next move: " ++ show selectedMove

```

```

66         putStrLn "Updated board state:"
67         print updatedBoard
68
69 -- Parallel Minimax with Alpha-Beta Pruning and controlled parallel depth
70 minimax :: Board -> Player -> Int -> Int -> Int -> Int -> Int
71 minimax board player depth alpha beta parallelDepth
72     | depth == 0 || null possibleMoves = evaluateBoard board player -- Base case
73     | depth > parallelDepth = sequentialMinimax alpha beta possibleMoves
74     | depth == parallelDepth = parallelMinimax alpha beta possibleMoves
75     | otherwise = sequentialMinimax alpha beta possibleMoves
76 where
77     possibleMoves = movesForPlayer board player
78
79 -- Sequential minimax
80 sequentialMinimax :: Int -> Int -> [[Coordinate]] -> Int
81 sequentialMinimax a b [] = if player == maximizingPlayer then a else b
82 sequentialMinimax a b (move:moves)
83     | player == maximizingPlayer = maximizing a b moves move
84     | otherwise = minimizing a b moves move
85 where
86     maximizing a b moves move = sequentialMinimax (max a (minimax (applyMove board
87 player move) (opposingPlayer player) (depth - 1) a b parallelDepth)) b moves
87     minimizing a b moves move = sequentialMinimax a (min b (minimax (applyMove
88 board player move) (opposingPlayer player) (depth - 1) a b parallelDepth)) moves
88
89 -- Parallel minimax
90 parallelMinimax :: Int -> Int -> [[Coordinate]] -> Int
91 parallelMinimax a b moves =
92     let results = map evaluateMove moves `using` parBuffer 2 rdeepseq
93     in if player == maximizingPlayer
94         then maximum results
95         else minimum results
96 where
97     evaluateMove move = minimax (applyMove board player move) (opposingPlayer
98 player) (depth - 1) a b parallelDepth
99
100     maximizingPlayer = Red -- Assume Red is the maximizing player
101
102 -- AI with parallelism and controlled depth
103 getAIMoveParallel :: GameSetup -> Player -> Int -> Int -> [Coordinate]
104 getAIMoveParallel setup player depth parallelDepth
105     | null possibleMoves = [] -- No moves available
106     | otherwise = fst $ maximumBy (comparing snd) evaluatedMoves
107 where
108     -- Get all possible moves for the current player
109     possibleMoves = movesForPlayer (board setup) player
110
111     -- Evaluate moves in parallel
112     evaluatedMoves = map evaluateMove possibleMoves `using` parBuffer 2 rdeepseq
113     evaluateMove move = (move, minimax (applyMove (board setup) player move)
114 (opposingPlayer player) (depth - 1) minBound maxBound parallelDepth) `using` rdeepseq

```

```

113
114 evaluateBoard :: Board -> Player -> Int
115 evaluateBoard board player = scoreForPlayer - scoreForOpponent
116   where
117     scoreForPlayer = length [pos | (pos, state) <- assocs (grid board), state == Placed
player]
118     scoreForOpponent = length [pos | (pos, state) <- assocs (grid board), state ==
Placed (opposingPlayer player)]
119

```

othello/app/side_side_7.hs

```

1  module Main where
2
3
4  import Data.List.Split (chunksOf)
5  import GHC.Conc (numCapabilities)
6  import System.Environment (getArgs)
7  import System.IO (readFile)
8  import Data.Array (array, listArray, assocs)
9  import Control.Parallel.Strategies (parMap, rdeepseq, parBuffer, using)
10 import Data.List (maximumBy)
11 import Data.Ord (comparing)
12 import Othello.GameLogic (
13     GameSetup(..), Player(..), initialGameState, Board(..), DiscState(..),
14     movesForPlayer, applyMove, Coordinate, opposingPlayer
15 )
16
17 -- Parse a custom board state from a string
18 parseCustomBoard :: String -> Int -> Board
19 parseCustomBoard input dim = Board {
20     grid = listArray ((0, 0), (dim-1, dim-1)) discStates,
21     boardDim = dim
22 }
23 where
24     rows = lines input
25     discStates = concatMap parseRow rows
26     parseRow row = map parseDiscState (words row)
27     parseDiscState "E" = Empty
28     parseDiscState "R" = Placed Red
29     parseDiscState "B" = Placed Blue
30     parseDiscState _ = error "Invalid disc state in custom board. Use 'E', 'R', or
'B'."
31
32 -- Define a simple entry point for the program
33 -- Define a simple entry point for the program
34 main :: IO ()
35 main = do
36     -- Get command-line arguments
37     args <- getArgs
38

```

```

39 -- Parse depth arguments
40 let (depth, parallelDepth, inputFile) = case args of
41     (d:p:file:_) -> (read d, read p, Just file)
42     (d:p:_)      -> (read d, read p, Nothing)
43     _            -> error "Usage: <depth> <parallelDepth> [gameboard file]"
44
45 -- Load the custom board or use the default initial game state
46 gameSetup <- case inputFile of
47     Just file -> do
48         content <- readFile file
49         let customBoard = parseCustomBoard content 8 -- Assume an 8x8 board for
simplicity
50         return GameSetup {
51             board = customBoard,
52             aiPlays = [Red, Blue],
53             searchDepth = depth
54         }
55     Nothing -> return (initialGameState 8 [Red, Blue] depth)
56
57 putStrLn $ "Using minimax depth: " ++ show depth
58 putStrLn $ "Parallelizing at depth: " ++ show parallelDepth
59
60 let currentPlayer = Red
61
62 -- Calculate the next move for the current player with the specified depth
63 let moves = getAIMoveParallel gameSetup currentPlayer depth parallelDepth
64 if null moves
65     then putStrLn "No valid moves available."
66     else do
67         let selectedMove = head moves
68         let updatedBoard = applyMove (board gameSetup) currentPlayer [selectedMove]
69         putStrLn $ "Next move: " ++ show selectedMove
70         putStrLn "Updated board state:"
71         print updatedBoard
72
73 --minimax search tree
74 minimax :: Board -> Player -> Int -> Int -> Int -> Int -> Int
75 minimax board player depth alpha beta parallelDepth
76     | depth == 0 || null possibleMoves = evaluateBoard board player -- Base case
77     | depth > parallelDepth = sequentialMinimax alpha beta possibleMoves
78     | depth == parallelDepth = parallelMinimax alpha beta possibleMoves
79     | otherwise = sequentialMinimax alpha beta possibleMoves
80 where
81     possibleMoves = movesForPlayer board player
82
83 -- Sequential minimax
84 sequentialMinimax :: Int -> Int -> [[Coordinate]] -> Int
85 sequentialMinimax a b [] = if player == maximizingPlayer then a else b
86 sequentialMinimax a b (move:moves)
87     | player == maximizingPlayer = maximizing a b moves move

```

```

88     | otherwise = minimizing a b moves move
89     where
90         maximizing a b moves move = sequentialMinimax (max a (minimax (applyMove board
player move) (opposingPlayer player) (depth - 1) a b parallelDepth)) b moves
91         minimizing a b moves move = sequentialMinimax a (min b (minimax (applyMove
board player move) (opposingPlayer player) (depth - 1) a b parallelDepth)) moves
92
93     -- Parallel minimax
94     parallelMinimax :: Int -> Int -> [[Coordinate]] -> Int
95     parallelMinimax a b moves =
96         let
97             -- Number of threads available
98             threads = numCapabilities
99
100            bufferSize = max 1 (round (2 * fromIntegral threads))
101            chunkSize = max 1 (length moves `div` (max 1 (round (fromIntegral threads *
fromIntegral threads/3))))
102            chunkedMoves = chunksOf chunkSize moves
103
104            results = map (maximum . map evaluateMove) chunkedMoves `using` parBuffer
bufferSize rdeepseq
105            in if player == maximizingPlayer
106                then maximum results
107                else minimum results
108        where
109            -- Evaluates a single move
110            evaluateMove move = minimax (applyMove board player move) (opposingPlayer
player) (depth - 1) a b parallelDepth
111            maximizingPlayer = Red -- Assume Red is the maximizing player
112
113    -- AI with parallelism and controlled depth
114    getAIMoveParallel :: GameSetup -> Player -> Int -> Int -> [Coordinate]
115    getAIMoveParallel setup player depth parallelDepth
116        | null possibleMoves = [] -- No moves available
117        | otherwise = fst $ maximumBy (comparing snd) evaluatedMoves
118    where
119        -- Get all possible moves for the current player
120        possibleMoves = movesForPlayer (board setup) player
121
122        -- Evaluate moves in parallel
123        evaluatedMoves = map evaluateMove possibleMoves `using` parBuffer 2 rdeepseq
124        evaluateMove move = (move, minimax (applyMove (board setup) player move)
(opposingPlayer player) (depth - 1) minBound maxBound parallelDepth) `using` rdeepseq
125
126    evaluateBoard :: Board -> Player -> Int
127    evaluateBoard board player = scoreForPlayer - scoreForOpponent
128    where
129        scoreForPlayer = length [pos | (pos, state) <- assocs (grid board), state == Placed
player]
130        scoreForOpponent = length [pos | (pos, state) <- assocs (grid board), state ==
Placed (opposingPlayer player)]

```

othello/app/side_side_8.hs

```

1  module Main where
2
3  import System.Environment (getArgs)
4  import System.IO (readFile)
5  import Data.Array (array, listArray, assocs)
6  import Data.Foldable (maximumBy)
7  import Data.Ord (comparing)
8  import Control.Parallel (par, pseq)
9  import Control.Parallel.Strategies (parList, parListChunk, rdeepseq, using, parMap)
10 import Othello.GameLogic (
11     GameSetup(..), Player(..), initialState, Board(..), DiscState(..),
12     movesForPlayer, applyMove, Coordinate, opposingPlayer
13 )
14
15 -- Parse a custom board state from a string
16 parseCustomBoard :: String -> Int -> Board
17 parseCustomBoard input dim = Board {
18     grid = listArray ((0, 0), (dim-1, dim-1)) discStates,
19     boardDim = dim
20 }
21 where
22     rows = lines input
23     discStates = concatMap parseRow rows
24     parseRow row = map parseDiscState (words row)
25     parseDiscState "E" = Empty
26     parseDiscState "R" = Placed Red
27     parseDiscState "B" = Placed Blue
28     parseDiscState _ = error "Invalid disc state in custom board. Use 'E', 'R', or
29 'B'."
30
31 -- Define a simple entry point for the program
32 main :: IO ()
33 main = do
34     -- Get command-line arguments
35     args <- getArgs
36
37     -- Parse depth arguments
38     let (depth, parallelDepth, inputFile) = case args of
39         (d:p:file:_) -> (read d, read p, Just file)
40         (d:p:_)      -> (read d, read p, Nothing)
41         _            -> error "Usage: <depth> <parallelDepth> [gameboard file]"
42
43     -- Load the custom board or use the default initial game state
44     gameSetup <- case inputFile of
45         Just file -> do
46             content <- readFile file

```

```

46      let customBoard = parseCustomBoard content 8 -- Assume an 8x8 board for
simplicity
47      return GameSetup {
48          board = customBoard,
49          aiPlays = [Red, Blue],
50          searchDepth = depth
51      }
52      Nothing -> return (initialGameState 8 [Red, Blue] depth)
53
54  putStrLn $ "Using minimax depth: " ++ show depth
55  putStrLn $ "Parallelizing at depth: " ++ show parallelDepth
56
57  let currentPlayer = Red
58
59  -- Calculate the next move for the current player with the specified depth
60  let moves = getAIMove gameSetup currentPlayer depth parallelDepth
61  if null moves
62      then putStrLn "No valid moves available."
63      else do
64          let selectedMove = head moves
65              let updatedBoard = applyMove (board gameSetup) currentPlayer [selectedMove]
66                  putStrLn $ "Next move: " ++ show selectedMove
67                  putStrLn "Updated board state:"
68                  print updatedBoard
69
70  -- Minimax with optional parallelization
71  minimax :: Board -> Player -> Int -> Int -> Int -> Int -> Int
72  minimax board player depth alpha beta parallelDepth
73      | depth == 0 || null possibleMoves = evaluateBoard board player -- Base case:
evaluate board
74      | depth == parallelDepth = parallelMinimax possibleMoves alpha beta
75      | otherwise = sequentialMinimax possibleMoves alpha beta
76  where
77      possibleMoves = movesForPlayer board player
78      maximizingPlayer = Red -- Assume Red is the maximizing player
79
80  -- Parallel minimax evaluation (Option 1: Parallelize at one depth)
81  parallelMinimax :: [[Coordinate]] -> Int -> Int -> Int
82  parallelMinimax moves a b =
83      let results = parMap rdeepseq (evaluateMove board player (depth - 1) a b) moves
84          in maximum results
85
86  -- Sequential minimax evaluation
87  sequentialMinimax :: [[Coordinate]] -> Int -> Int -> Int
88  sequentialMinimax [] a _ = a -- No more moves, return alpha
89  sequentialMinimax (move:moves) a b
90      | a >= b = a -- Prune the rest of the tree
91      | otherwise = sequentialMinimax moves a' b
92  where
93      a' = max a (evaluateMove board player (depth - 1) a b move)

```

```

94
95     evaluateMove :: Board -> Player -> Int -> Int -> Int -> [Coordinate] -> Int
96     evaluateMove b p d a b' move = minimax (applyMove b p move) (opposingPlayer p) d a
b' parallelDepth
97
98 -- Parallelization Option 2: Divide the Tree Aggressively
99 parallelMinimaxAggressive :: Board -> Player -> Int -> Int -> Int -> [[Coordinate]] ->
Int
100 parallelMinimaxAggressive board player depth a b moves =
101     let chunkedMoves = map (:[]) moves `using` parListChunk 2 rdeepseq
102         results = map (maximum . map (evaluateMove board player (depth - 1) a b))
chunkedMoves
103     in maximum results
104     where
105         evaluateMove :: Board -> Player -> Int -> Int -> Int -> [Coordinate] -> Int
106         evaluateMove b p d a b' move = minimax (applyMove b p move) (opposingPlayer p) d a
b' depth
107
108 -- AI
109 getAIMove :: GameSetup -> Player -> Int -> Int -> [Coordinate]
110 getAIMove setup player depth parallelDepth
111     | null possibleMoves = [] -- No moves available
112     | otherwise = fst $ maximumBy (comparing snd) evaluatedMoves
113     where
114         -- Get all possible moves for the current player
115         possibleMoves = movesForPlayer (board setup) player
116
117         -- Evaluate each move using minimax
118         evaluatedMoves = [
119             (move, minimax (applyMove (board setup) player move) (opposingPlayer player)
(depth - 1) minBound maxBound parallelDepth)
120             | move <- possibleMoves
121         ]
122
123 -- Example board evaluation function
124 evaluateBoard :: Board -> Player -> Int
125 evaluateBoard board player = scoreForPlayer - scoreForOpponent
126     where
127         scoreForPlayer = length [pos | (pos, state) <- assocs (grid board), state == Placed
player]
128         scoreForOpponent = length [pos | (pos, state) <- assocs (grid board), state ==
Placed (opposingPlayer player)]
129

```

othello/app/side_side_9.hs

```

1 module Main where
2
3 import GHC.Conc (numCapabilities)
4 import System.Environment (getArgs)
5 import System.IO (readFile)

```



```

6 import Data.Array (array, listArray, assocs)
7 import Data.Foldable (maximumBy)
8 import Data.Ord (comparing)
9 import Control.Parallel (par, pseq)
10 import Control.Parallel.Strategies (parList, parListChunk, rdeepseq, using, parMap)
11 import Othello.GameLogic (
12     GameSetup(..), Player(..), initialGameState, Board(..), DiscState(..),
13     movesForPlayer, applyMove, Coordinate, opposingPlayer
14 )
15
16 -- Parse a custom board state from a string
17 parseCustomBoard :: String -> Int -> Board
18 parseCustomBoard input dim = Board {
19     grid = listArray ((0, 0), (dim-1, dim-1)) discStates,
20     boardDim = dim
21 }
22 where
23     rows = lines input
24     discStates = concatMap parseRow rows
25     parseRow row = map parseDiscState (words row)
26     parseDiscState "E" = Empty
27     parseDiscState "R" = Placed Red
28     parseDiscState "B" = Placed Blue
29     parseDiscState _ = error "Invalid disc state in custom board. Use 'E', 'R', or
30 'B'."
31
32 -- Define a simple entry point for the program
33 main :: IO ()
34 main = do
35     -- Get command-line arguments
36     args <- getArgs
37
38     -- Parse depth arguments
39     let (depth, parallelDepth, inputFile) = case args of
40         (d:p:file:_) -> (read d, read p, Just file)
41         (d:p:_)      -> (read d, read p, Nothing)
42         _            -> error "Usage: <depth> <parallelDepth> [gameboard file]"
43
44     -- Load the custom board or use the default initial game state
45     gameSetup <- case inputFile of
46         Just file -> do
47             content <- readFile file
48             let customBoard = parseCustomBoard content 8 -- Assume an 8x8 board for
49                 simplicity
50                 return GameSetup {
51                     board = customBoard,
52                     aiPlays = [Red, Blue],
53                     searchDepth = depth
54                 }
55         Nothing -> return (initialGameState 8 [Red, Blue] depth)

```

```

54
55 putStrLn $ "Using minimax depth: " ++ show depth
56 putStrLn $ "Parallelizing at depth: " ++ show parallelDepth
57
58 let currentPlayer = Red
59
60 -- Calculate the next move for the current player with the specified depth
61 let moves = getAIMove gameSetup currentPlayer depth parallelDepth
62 if null moves
63     then putStrLn "No valid moves available."
64     else do
65         let selectedMove = head moves
66             updatedBoard = applyMove (board gameSetup) currentPlayer [selectedMove]
67             putStrLn $ "Next move: " ++ show selectedMove
68             putStrLn "Updated board state:"
69             print updatedBoard
70
71 -- Minimax game board
72 minimax :: Board -> Player -> Int -> Int -> Int -> Int -> Int
73 minimax board player depth alpha beta parallelDepth
74     | depth == 0 || null possibleMoves = evaluateBoard board player -- Base case
75     | depth == parallelDepth = parallelMinimax possibleMoves alpha beta
76     | otherwise = sequentialMinimax possibleMoves alpha beta
77 where
78     possibleMoves = movesForPlayer board player
79     maximizingPlayer = Red -- Assume Red is the maximizing player
80
81 -- Parallel minimax evaluation
82 parallelMinimax :: [[Coordinate]] -> Int -> Int -> Int
83 parallelMinimax moves a b =
84     let results = parMap rdeepseq (evaluateMove board player (depth - 1) a b) moves
85     in maximum results
86
87 -- Sequential minimax evaluation
88 sequentialMinimax :: [[Coordinate]] -> Int -> Int -> Int
89 sequentialMinimax [] a _ = a -- No more moves, return alpha
90 sequentialMinimax (move:moves) a b
91     | a >= b = a -- Prune the rest of the tree
92     | otherwise = sequentialMinimax moves a' b
93 where
94     a' = max a (evaluateMove board player (depth - 1) a b move)
95
96 evaluateMove :: Board -> Player -> Int -> Int -> Int -> [Coordinate] -> Int
97 evaluateMove b p d a b' move = minimax (applyMove b p move) (opposingPlayer p) d a
98 b' parallelDepth
99
100 -- AI
101 getAIMove :: GameSetup -> Player -> Int -> Int -> [Coordinate]
102 getAIMove setup player depth parallelDepth
103     | null possibleMoves = [] -- No moves available

```

```

103 | otherwise = fst $ maximumBy (comparing snd) evaluatedMoves
104 | where
105 |   -- Get all possible moves for the current player
106 |   possibleMoves = movesForPlayer (board setup) player
107 |
108 |   -- Evaluate each move using minimax
109 |   evaluatedMoves = [
110 |     (move, minimax (applyMove (board setup) player move) (opposingPlayer player)
111 |       (depth - 1) minBound maxBound parallelDepth)
112 |     | move <- possibleMoves
113 |   ]
114 | --evaluate Board
115 | evaluateBoard :: Board -> Player -> Int
116 | evaluateBoard board player = scoreForPlayer - scoreForOpponent
117 |   where
118 |     scoreForPlayer = length [pos | (pos, state) <- assocs (grid board), state == Placed
119 |       player]
120 |     scoreForOpponent = length [pos | (pos, state) <- assocs (grid board), state ==
121 |       Placed (opposingPlayer player)]

```

othello/benchmark_performance.py

```

1 | import subprocess
2 | import sys
3 | import re
4 | import argparse
5 |
6 | def parse_output(stdout):
7 |     total_time_pattern = r"Total\s+time\s+\d+\.\d+s\s+\(\s+([\d\.]+)\s+"
8 |     tasks_pattern = r"TASKS:\s+(\d+)"
9 |     sparks_pattern = r"SPARKS:\s+(\d+)"
10 |
11 |     total_time_match = re.search(total_time_pattern, stdout)
12 |     tasks_match = re.search(tasks_pattern, stdout)
13 |     sparks_match = re.search(sparks_pattern, stdout)
14 |
15 |     total_time = float(total_time_match.group(1)) if total_time_match else None
16 |     tasks = int(tasks_match.group(1)) if tasks_match else None
17 |     sparks = int(sparks_match.group(1)) if sparks_match else None
18 |
19 |     return total_time, tasks, sparks
20 |
21 | def run_benchmark(test_file="custom_game_1.txt", iterate="parallel_depth", depth=7,
22 | parallel_depth=1):
23 |     results = {}
24 |     iteration_range = range(1, max(depth, parallel_depth) + 1)
25 |     threads_range = range(1, 9)
26 |
27 |     if iterate == "depth":
28 |         depth_range = iteration_range

```

```

28     parallel_depth_range = [parallel_depth]
29     threads_range = [4]
30     elif iterate == "parallel_depth":
31         depth_range = [depth]
32         parallel_depth_range = iteration_range
33         threads_range = [4]
34     elif iterate == "num_threads":
35         depth_range = [depth]
36         parallel_depth_range = [parallel_depth]
37     else:
38         print(f"Invalid iterate value: {iterate}. Use 'depth', 'parallel_depth', or
'num_threads'.")
39         return
40
41     for d in depth_range:
42         for pd in parallel_depth_range:
43             for threads in threads_range:
44                 print(f"Running tests with depth {d}, parallel depth {pd}, threads
{threads}...")
45
46                 total_times = []
47                 total_tasks = []
48                 total_sparks = []
49
50                 for _ in range(5):
51                     # Prepare the command
52                     command = [
53                         "cabal", "run", "othello", "--", str(d), str(pd), test_file,
"+RTS", "-s", f"-N{threads}"
54                     ]
55
56                 try:
57                     result = subprocess.run(command, capture_output=True,
text=True, check=True)
58                     stdout = result.stdout
59                     stderr = result.stderr
60
61                     total_time, tasks, sparks = parse_output(stdout + stderr)
62
63                     if total_time is not None and tasks is not None and sparks is
not None:
64                         total_times.append(total_time)
65                         total_tasks.append(tasks)
66                         total_sparks.append(sparks)
67                     else:
68                         print(f"Warning: Missing data in the output for depth {d},
parallel depth {pd}, threads {threads}")
69
70
71                 except subprocess.CalledProcessError as e:
72                     print(f"Error running command {command}: {e}")

```

```

73         continue
74
75     # Compute averages for this configuration
76     if total_times:
77         avg_total_time = sum(total_times) / len(total_times)
78         avg_tasks = sum(total_tasks) / len(total_tasks)
79         avg_sparks = sum(total_sparks) / len(total_sparks)
80         results[(d, pd, threads)] = {
81             "avg_total_time": avg_total_time,
82             "avg_tasks": avg_tasks,
83             "avg_sparks": avg_sparks
84         }
85
86     for (d, pd, threads), data in results.items():
87         print(f"Depth {d}, Parallel depth {pd}, Threads {threads}:")
88         print(f"  Average Total Time: {data['avg_total_time']:.3f}s")
89         print(f"  Average Tasks: {data['avg_tasks']}")
90         print(f"  Average Sparks: {data['avg_sparks']}")
91
92 if __name__ == "__main__":
93     parser = argparse.ArgumentParser(description="Run benchmarks for Othello.")
94     parser.add_argument("--test_file", type=str, default="custom_game_1.txt",
95 help="Path to the test file.")
96     parser.add_argument("--iterate", type=str, choices=["depth", "parallel_depth",
97 "num_threads"], default="parallel_depth",
98 help="Parameter to iterate over: 'depth', 'parallel_depth', or
99 'num_threads'.")
100     parser.add_argument("--depth", type=int, default=7, help="Initial depth value.")
101     parser.add_argument("--parallel_depth", type=int, default=1, help="Initial parallel
102 depth value.")
103
104     args = parser.parse_args()
105
106     run_benchmark(
107         test_file=args.test_file,
108         iterate=args.iterate,
109         depth=args.depth,
110         parallel_depth=args.parallel_depth
111     )

```

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

```
1 cabal-version:      2.4
2 name:               haskell-othello
3 version:            0.1.0.0
4
5 -- A short (one-line) description of the package.
6 synopsis:           Experiments in parallelizing an Othello minmax agent.
7
8 -- A longer description of the package.
9 description:        Experiments in parallelizing an Othello minmax agent.
10
11 -- A URL where users can report bugs.
12 bug-reports:       https://github.com/NoamHirschorn/PFP\_final/issues
13
14 -- The license under which the package is released.
15 --license:         MIT
16 --license-file:    LICENSE
17
18 -- The package author(s).
19 --author:          Noam Hirschorn, Dan Ivanovich. Adapted from code by Arttu. Y
20
21 -- An email address to which users can send suggestions, bug reports, and patches.
22 --maintainer:      dmi2115@columbia.edu
23
24 -- A copyright notice.
25 copyright:         2024 Noam Hirschorn, Dan Ivanovich, Arttu. Y
26 category:          Games
```

```

27
28 -- extra-source-files: CHANGELOG.md
29
30 library
31     exposed-modules:  Othello.GameLogic
32
33     -- Modules included in this library but not exported.
34     -- other-modules:
35
36     -- LANGUAGE extensions used by modules in this package.
37     -- other-extensions:
38
39     build-depends:
40         base >=4.14.3.0 && <4.19,
41         array,
42         parallel,
43         deepseq >= 1.4,
44         split
45
46     hs-source-dirs:  src
47     default-language: Haskell2010
48
49 executable othello
50     main-is:          side_side_9.hs
51
52     -- Modules included in this executable, other than Main.
53     -- other-modules:
54
55     -- LANGUAGE extensions used by modules in this package.
56     -- other-extensions:
57
58     build-depends:
59         base >=4.14.3.0 && <4.19,
60         array,
61         parallel,
62         deepseq >= 1.4,
63         split,
64         haskell-othello
65
66     hs-source-dirs:  app
67     default-language: Haskell2010
68
69     ghc-options:      -threaded -rtsopts -with-rtsopts=-N -debug
70

```

othello/src/Othello/GameLogic.hs

```

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

```

```

4
5 module Othello.GameLogic where
6 import Data.Array ( Array, array, elems, inRange, bounds, (//) )
7 import qualified Data.Array ((!))
8 import Data.Foldable (maximumBy, minimumBy)
9 import Data.Ord (comparing)
10 import Data.Array (Array, assocs, elems, bounds, (//))
11 -- Score is an integer, and coordinate is a pair of integers
12 type UnitScore = Int
13 type Coordinate = (Int, Int)
14
15 -- A player is either Red, or Blue. Derive comparison and Show
16 data Player = Red | Blue deriving (Eq, Show)
17
18 -- Each spot on a board is either empty, or placed with some player
19 data DiscState = Empty | Placed Player deriving (Eq, Show)
20
21 -- Board is essentially a grid of disc states, with the size attached
22 data Board = Board {
23     grid :: Array Coordinate DiscState,
24     boardDim :: Int
25 } deriving (Show)
26
27 -- For purposes of Minimax AI, we will need to measure the score of a given state. It
28 -- will be either a win, indeterminate with score of some kind (from the view of who is
29 -- requesting it), or a loss
30 data BoardScore = Win | Indeterminate UnitScore | Lose deriving (Eq, Show)
31
32 -- Define a order for a board score. For least complexity, define ordering as a set of
33 -- comparative properties between different scores
34 instance Ord BoardScore where
35     (<=) :: BoardScore -> BoardScore -> Bool
36     (<=) Lose _ = True -- Lose is the smallest and definitely equal
37     (<=) (Indeterminate _) Lose = False -- Indeterminate is never less or equal to a
38     win
39     (<=) (Indeterminate _) Win = True -- Indeterminate is always less than a win
40     (<=) (Indeterminate a) (Indeterminate b) = a <= b -- For two indeterminates, their
41     respective ordering depends on their scores
42     (<=) Win Win = True -- Win is equal with a win
43     (<=) Win _ = False -- Otherwise, no
44
45 -- Define a scoring function; given a player and a board, what is their score?
46 score :: Player -> Board -> BoardScore
47 score player board
48     | not (movesPossibleOnBoard board) && redLeading = if player == Red then Win else
49     Lose
50     | not (movesPossibleOnBoard board) && blueLeading = if player == Blue then Win else
51     Lose
52     | otherwise = Indeterminate (if player == Red then redCount else blueCount)
53 where
54     redLeading = redCount > blueCount

```



```

48     blueLeading = blueCount > redCount
49
50     (redCount, blueCount) = pieceCount board
51
52 data GameSetup = GameSetup {
53     board :: Board, -- Board
54     aiPlays :: [Player], -- Which turns AI plays?
55     searchDepth :: Int -- Search depth
56 } deriving (Show)
57 -- Core functions
58
59 -- Count of pieces, per color, for the board
60 pieceCount :: Board -> (Int, Int)
61 pieceCount board = foldr adder (0,0) (elems $ grid board) -- Add element by element,
start with zero scores for both
62     where
63         adder :: DiscState -> (Int, Int) -> (Int, Int)
64         adder state count@(red, blue) = case state of
65             Empty -> count
66             Placed Red -> (red+1, blue)
67             Placed Blue -> (red, blue+1)
68
69 -- Definition of the opposing player for a given player
70 opposingPlayer :: Player -> Player
71 opposingPlayer Red = Blue
72 opposingPlayer Blue = Red
73
74 -- Calculating which player is winning by their score; this does not consider if there
are more turns remaining
75 playerWithBestScore :: Board -> Maybe Player
76 playerWithBestScore board
77     | red == blue = Nothing
78     | otherwise = if red > blue then Just Red else Just Blue
79     where
80         (red, blue) = pieceCount board
81
82 -- Define an indexing operation for a board, quite alike what Arrays have
83 (!) :: Board -> Coordinate -> DiscState
84 board ! coordinate = (Data.Array.!) (grid board) coordinate
85
86 -- Define a validity check operator for indexes; this will return true if the index is
acceptable
87 (!?) :: Board -> Coordinate -> Bool
88 board !? coord = inRange (bounds $ grid board) coord
89
90 -- Define a grid comprehension function; mapping over coordinates of a grid, construct
some array of data
91 comprehensionByBoard :: Board -> (Coordinate -> x) -> [x]
92 comprehensionByBoard board = comprehensionByDim size
93     where
94         size = boardDim board

```

```

95
96 comprehensionByDim :: Int -> (Coordinate -> x) -> [x]
97 comprehensionByDim size func = [ func (a,b) | a <- [0..size-1], b <- [0..size-1]]
98
99 initialGameState :: Int -> [Player] -> Int -> GameSetup
100 initialGameState dim aiPlays searchDepth = GameSetup { aiPlays, searchDepth, board }
101   where
102     board = Board { boardDim = dim, grid = array ((0, 0), (dim-1,dim-1))
103 (comprehensionByDim dim (\p -> (p, startPieces p))) }
104
105     startPieces :: Coordinate -> DiscState
106     startPieces (cx, cy)
107       | cx == (dim `div` 2) - 1 && cy == (dim `div` 2) - 1 = Placed Blue
108       | cx == (dim `div` 2) && cy == (dim `div` 2) = Placed Blue
109       | cx == (dim `div` 2) - 1 && cy == (dim `div` 2) = Placed Red
110       | cx == (dim `div` 2) && cy == (dim `div` 2) - 1 = Placed Red
111       | otherwise = Empty
112
113 ----- Moves and AI
114
115 -- For a given board, player and coordinate, determine what coordinates should be
116 -- changed to player's color. If an empty list is returned, move is not valid
117 -- This also includes the starting point given
118 getMovesOnPoint :: Board -> Player -> Coordinate -> [Coordinate]
119 getMovesOnPoint board player startCoord@(sx, sy)
120   | not (board !? startCoord) = [] -- Coordinate is not a valid position
121   | startPiece /= Empty = [] -- Starting piece is not empty
122   | null resultSteps = [] -- No valid steps exist
123   | otherwise = startCoord : resultSteps
124 where
125   resultSteps = concatMap walkAndMark directions -- Valid steps are a
126 concatenation of walked directions - per rules, we can and must mark all branched paths
127
128   walkAndMark :: Coordinate -> [Coordinate] -- If stepping from a given
129 direction, what can we mark (excluding start position)?
130   walkAndMark dir@(dx, dy) = walkAndMark' (sx+dx, sy+dy) dir []
131
132   walkAndMark' :: Coordinate -> Coordinate -> [Coordinate] -> [Coordinate] --
133 Current position, direction, found already
134   walkAndMark' cur@(cx, cy) dir@(dx, dy) found
135   | isEndPiece = found -- End piece, terminate here
136   | isValidTraversalPiece = walkAndMark' (cx+dx, cy+dy) dir (cur:found) --
137 Can traverse, step forward
138   | otherwise = [] -- No valid way to travel nor end, return nothing
139 where
140   isEndPiece = isValidPos && board ! cur == Placed player
141   isValidTraversalPiece = isValidPos && board ! cur == Placed
142 (opposingPlayer player)
143   isValidPos = board !? cur
144
145   directions = [(-1,-1), (-1,0), (-1,1), (0,-1), (0,1), (1,-1), (1,0), (1,1)] --
146 Which directions need to be checked. As defined in the rules, we work on either

```

```

horizontal, vertical or diagonal lines
138     startPiece = board ! startCoord
139
140 movesForPlayer :: Board -> Player -> [[Coordinate]] -- Return all movesets that are
nonempty, for a given player?
141 movesForPlayer board player = filter (not . null) mappedCoords
142     where
143         mappedCoords = comprehensionByBoard board (getMovesOnPoint board player)
144
145 movesPossibleOnBoard :: Board -> Bool -- Are there any moves possible on board?
146 movesPossibleOnBoard board = not (null (movesForPlayer board Red) && null
(movesForPlayer board Blue))
147
148 applyMove :: Board -> Player -> [Coordinate] -> Board
149 applyMove board player moveList = board { grid = grid board // map (, Placed player)
moveList }
150
151

```

othello/src/Othello/backup_GameLog.hs

```

1 {-#LANGUAGE InstanceSigs#-} -- Permit type declarations in instance definitions
2 {-# LANGUAGE TupleSections #-} -- Partial tuple constructors as functions
3 {-# LANGUAGE NamedFieldPuns #-} -- Allow more elegant construction of data
4
5 module Othello.GameLogic where
6 import Data.Array ( Array, array, elems, inRange, bounds, (//) )
7 import qualified Data.Array ((!))
8 import Data.Foldable (maximumBy, minimumBy)
9 import Data.Ord (comparing)
10 import Data.Array (Array, assocs, elems, bounds, (//))
11 -- Score is an integer, and coordinate is a pair of integers
12 type UnitScore = Int
13 type Coordinate = (Int, Int)
14
15 -- A player is either Red, or Blue. Derive comparison and Show
16 data Player = Red | Blue deriving (Eq, Show)
17
18 -- Each spot on a board is either empty, or placed with some player
19 data DiscState = Empty | Placed Player deriving (Eq, Show)
20
21 -- Board is essentially a grid of disc states, with the size attached
22 data Board = Board {
23     grid :: Array Coordinate DiscState,
24     boardDim :: Int
25 } deriving (Show)
26
27 -- 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
28 data BoardScore = Win | Indeterminate UnitScore | Lose deriving (Eq, Show)

```

```

29
30 -- Define a order for a board score. For least complexity, define ordering as a set of
comparative properties between different scores
31 instance Ord BoardScore where
32     (<=) :: BoardScore -> BoardScore -> Bool
33     (<=) Lose _ = True -- Lose is the smallest and definitely equal
34     (<=) (Indeterminate _) Lose = False -- Indeterminate is never less or equal to a
win
35     (<=) (Indeterminate _) Win = True -- Indeterminate is always less than a win
36     (<=) (Indeterminate a) (Indeterminate b) = a <= b -- For two indeterminates, their
respective ordering depends on their scores
37     (<=) Win Win = True -- Win is equal with a win
38     (<=) Win _ = False -- Otherwise, no
39
40 -- Define a scoring function; given a player and a board, what is their score?
41 score :: Player -> Board -> BoardScore
42 score player board
43     | not (movesPossibleOnBoard board) && redLeading = if player == Red then Win else
Lose
44     | not (movesPossibleOnBoard board) && blueLeading = if player == Blue then Win else
Lose
45     | otherwise = Indeterminate (if player == Red then redCount else blueCount)
46     where
47         redLeading = redCount > blueCount
48         blueLeading = blueCount > redCount
49
50         (redCount, blueCount) = pieceCount board
51
52 data GameSetup = GameSetup {
53     board :: Board, -- Board
54     aiPlays :: [Player], -- Which turns AI plays?
55     searchDepth :: Int -- Search depth
56 } deriving (Show)
57 -- Core functions
58
59 -- Count of pieces, per color, for the board
60 pieceCount :: Board -> (Int, Int)
61 pieceCount board = foldr adder (0,0) (elems $ grid board) -- Add element by element,
start with zero scores for both
62     where
63         adder :: DiscState -> (Int, Int) -> (Int, Int)
64         adder state count@(red, blue) = case state of
65             Empty -> count
66             Placed Red -> (red+1, blue)
67             Placed Blue -> (red, blue+1)
68
69 -- Definition of the opposing player for a given player
70 opposingPlayer :: Player -> Player
71 opposingPlayer Red = Blue
72 opposingPlayer Blue = Red
73

```

```

74 -- Calculating which player is winning by their score; this does not consider if there
    are more turns remaining
75 playerWithBestScore :: Board -> Maybe Player
76 playerWithBestScore board
77     | red == blue = Nothing
78     | otherwise = if red > blue then Just Red else Just Blue
79     where
80         (red, blue) = pieceCount board
81
82 -- Define an indexing operation for a board, quite alike what Arrays have
83 (!) :: Board -> Coordinate -> DiscState
84 board ! coordinate = (Data.Array.!) (grid board) coordinate
85
86 -- Define a validity check operator for indexes; this will return true if the index is
    acceptable
87 (!?) :: Board -> Coordinate -> Bool
88 board !? coord = inRange (bounds $ grid board) coord
89
90 -- Define a grid comprehension function; mapping over coordinates of a grid, construct
    some array of data
91 comprehensionByBoard :: Board -> (Coordinate -> x) -> [x]
92 comprehensionByBoard board = comprehensionByDim size
93     where
94         size = boardDim board
95
96 comprehensionByDim :: Int -> (Coordinate -> x) -> [x]
97 comprehensionByDim size func = [ func (a,b) | a <- [0..size-1], b <- [0..size-1]]
98
99 initialGameState :: Int -> [Player] -> Int -> GameSetup
100 initialGameState dim aiPlays searchDepth = GameSetup { aiPlays, searchDepth, board }
101     where
102         board = Board { boardDim = dim, grid = array ((0, 0), (dim-1,dim-1))
    (comprehensionByDim dim (\p -> (p, startPieces p))) }
103
104         startPieces :: Coordinate -> DiscState
105         startPieces (cx, cy)
106             | cx == (dim `div` 2) - 1 && cy == (dim `div` 2) - 1 = Placed Blue
107             | cx == (dim `div` 2) && cy == (dim `div` 2) = Placed Blue
108             | cx == (dim `div` 2) - 1 && cy == (dim `div` 2) = Placed Red
109             | cx == (dim `div` 2) && cy == (dim `div` 2) - 1 = Placed Red
110             | otherwise = Empty
111 ---- Moves and AI
112
113 -- 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
114 -- This also includes the starting point given
115 getMovesOnPoint :: Board -> Player -> Coordinate -> [Coordinate]
116 getMovesOnPoint board player startCoord@(sx, sy)
117     | not (board !? startCoord) = [] -- Coordinate is not a valid position
118     | startPiece /= Empty = [] -- Starting piece is not empty
119     | null resultSteps = [] -- No valid steps exist

```

```

120 | otherwise = startCoord : resultSteps
121 where
122     resultSteps = concatMap walkAndMark directions -- Valid steps are a
concatenation of walked directions - per rules, we can and must mark all branched paths
123
124     walkAndMark :: Coordinate -> [Coordinate] -- If stepping from a given
direction, what can we mark (excluding start position)?
125     walkAndMark dir@(dx, dy) = walkAndMark' (sx+dx, sy+dy) dir []
126
127     walkAndMark' :: Coordinate -> Coordinate -> [Coordinate] -> [Coordinate] --
Current position, direction, found already
128     walkAndMark' cur@(cx, cy) dir@(dx, dy) found
129         | isEndPiece = found -- End piece, terminate here
130         | isValidTraversalPiece = walkAndMark' (cx+dx, cy+dy) dir (cur:found) --
Can traverse, step forward
131         | otherwise = [] -- No valid way to travel nor end, return nothing
132     where
133         isEndPiece = isValidPos && board ! cur == Placed player
134         isValidTraversalPiece = isValidPos && board ! cur == Placed
(opposingPlayer player)
135         isValidPos = board !? cur
136
137     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
138     startPiece = board ! startCoord
139
140 movesForPlayer :: Board -> Player -> [[Coordinate]] -- Return all movesets that are
nonempty, for a given player?
141 movesForPlayer board player = filter (not . null) mappedCoords
142     where
143         mappedCoords = comprehensionByBoard board (getMovesOnPoint board player)
144
145 movesPossibleOnBoard :: Board -> Bool -- Are there any moves possible on board?
146 movesPossibleOnBoard board = not (null (movesForPlayer board Red) && null
(movesForPlayer board Blue))
147
148 applyMove :: Board -> Player -> [Coordinate] -> Board
149 applyMove board player moveList = board { grid = grid board // map (, Placed player)
moveList }
150
151 -- AI
152
153 getAIMove :: GameSetup -> Player -> Int -> [Coordinate]
154 getAIMove setup player depth
155     | null possibleMoves = [] -- No moves available
156     | otherwise = fst $ maximumBy (comparing snd) evaluatedMoves
157 where
158     -- Get all possible moves for the current player
159     possibleMoves = movesForPlayer (board setup) player
160

```

```

161     -- Evaluate each move using minimax
162     evaluatedMoves = [(move, minimax (applyMove (board setup) player move)
    (opposingPlayer player) (depth - 1)) | move <- possibleMoves]
163
164 -- Minimax algorithm for game tree evaluation
165 minimax :: Board -> Player -> Int -> Int
166 minimax board player depth
167     | depth == 0 || null possibleMoves = evaluateBoard board player -- Base case:
    evaluate board
168     | player == maximizingPlayer = maximum [minimax (applyMove board player move)
    (opposingPlayer player) (depth - 1) | move <- possibleMoves]
169     | otherwise = minimum [minimax (applyMove board player move) (opposingPlayer
    player) (depth - 1) | move <- possibleMoves]
170     where
171         possibleMoves = movesForPlayer board player
172         maximizingPlayer = Red -- Assume Red is the maximizing player
173
174 -- Example board evaluation function
175 evaluateBoard :: Board -> Player -> Int
176 evaluateBoard board player = scoreForPlayer - scoreForOpponent
177     where
178         scoreForPlayer = length [pos | (pos, state) <- assocs (grid board), state == Placed
    player]
179         scoreForOpponent = length [pos | (pos, state) <- assocs (grid board), state ==
    Placed (opposingPlayer player)]
180

```