COMS 4995
Parallel Functional Programming

Backgammon with Parallelized Expectiminimax and Forward Pruning

Tejit Pabari (tvp2107)
Archit Choudhury (ac4385)

December 23, 2020

Introduction

Backgammon is a two-player game where each player has fifteen pieces that move between twenty-four triangles based on dice roll. The objective of the game is to be the first to move all your fifteen checkers off the board.

Detailed rules can be found at the [Wikipedia Page]

Problem Formulation

The game is similar to a 2 player zero-sum game. The possible moves at each step can be represented on a tree, based on the dice role and the current state of the board. The tree can be searched using an expecti minimax algorithm to find the best possible next move. A couple of heuristics can also be applied to improve calculations and optimize the algorithm, such as alpha-beta pruning and forward pruning. All these could be parallized, which would improve the performance of the algorithm.

Since the search tree is big, we can't find an exact answer. Again, quite similar to chess. So we could experiment with different tree depths, and the time it takes to perform the operations on 1 core. Following this, we can gradually increase the core count, and see how that affects the time taken to perform the same operation.

Implementation

Data Type Definitions

We used a couple of custom data types, which we would explain here

Die, Dice

Each Die is an Int and Dice is a pair of Die, i.e. (Die, Die)

Chip, Side

A chip is an Int, and denotes the number of pieces on each triangle of the board.
A side is either Black or White, denoting the side the player is playing on.

Point

A point is of Either type and is a tuple of (Side, Chip). Hence, it essentially denotes a triangle in the game board, with the number of chips of a given side present on it.
Note: This also defines an important part of the game that no triangle can have chips of two sides on it

Move

Move denotes a player's move on the board. It can be of 3 types, in order of their priorities

1. Enter - Enter your piece from the bar
2. BearOff - Bear off your piece, bringing you closer to winning
3. Move - Move your pieces in the board

If there are chips on the bar, BearOff is the only possible move. If you have a chance to BearOff at any point, you would definitely take it because it brings you closer to winning. Hence, these two specific move types have been separated.

Game State

At any point, the game can be in any of these states. Each state is followed by an action from the player (or AI)

1. ToMove Side Dice - Prompting player to Move from one position to another
2. ToThrow Dice - Prompting player to throw dice
3. GameFinished Side - Notes end of game, and what side won
4. PlayersToThrowInitial - Initial throws, which doesn't fit into any of the above types

Player Decision

Based on the game state, a player can decide to do these actions

1. Moves [Move] - A list of moves to move (you have multiple moves per turn)
2. Throw Dice - Throw a dice

Invalid Decision Type

These are just some error checks for invalid decisions done by the player

1. NoPieces Pos - Player moves from a position without pieces
2. MovedOntoOpponentsClosedPoint Pos - Player moved onto opponent's triangle (with more than 1 opponent's piece)
3. NoBarPieces Side - No bar pieces present

Game Action

Action during a game comprises a player decision. This is a wrapper for this. It also includes an initial throw action, which is not a regular player decision move.

Invalid Action

An action can either be invalid for a given state of the game, or it can be invalid because of a player decision taken or an invalid initial throw. This is also a wrapper for Player decision

Board

A board consists of Points, Int, Int - showing each triangle on the board, along with the pieces on bar for White, and pieces on bar for Black side respectively (the two ints)

Game

Finally, a game shows the game board, game action and game state at any given point. Hence, it is sort of a bookkeeping method to ensure correct gameplay throughout.

```
data Game = Game { gameBoard :: Board,
                   gameActions :: [GameAction],
                   gameState :: GameState}
```

Important Function Definitions

Move

Def ⇒ move :: Side -> Board -> Move -> Either InvalidDecisionType Board
Moves the pieces on the board, given the player side, board and single move
Returns the new board, or an invalid decision take by the player.
Handles all move types

Legal Moves

Def ⇒ legalMoves :: Board -> Dice -> Side -> [[Move]]
Calculates all possible legal moves of the player at a current point, given the board, the dice roll and the side of the player.
Order of calculation:

1. Bar moves
2. Bear Off moves
3. Other regular moves

Perform Action

Def ⇒ performAction :: GameAction -> Game -> Either InvalidAction Game
Given a game action (Initial Throw, Move, Throw), and a game (game state actually, it takes the game and extracts the current state from it), it matches the appropriate state with the action and performs the given action, or returns an invalid matching (for example, if game state is ToThrow and player says perform move action - that is invalid)

Game Play

Def ⇒ gamePlay :: Side -> p1 -> p2 -> Int -> {bestMove Func def} -> Either InvalidAction Game
These values are just passed on to the {bestMove Func}

- P1 is depth, P2 is pruning depth
- Int is the seed,
- {bestMove Func def} is the method of selecting the move (random or using expectiminimax)

Given a side, this function essentially auto plays the game.

Sequential Solution

Eval Function

Evaluates the current board, for the given side (player) and assigns a value to the board. This value is compared to select the best move for the player.
Eval = homeChips + 10 * homeWin - distance - 10 * barWeight - opponentChips where

- homeChips = number of chips on the home board
  (triangles 1..6 for black, triangles 19..24 for white)
- homeWin = number of chips that the player has BearedOff
- distance = weighted sum of how far each piece of the player is from being BearedOff
- barWeight = number of chips on the bar
- opponentChips = number of chips of the opponent present on their home board

Forward Pruning

Reduces the number of moves to k (some small number) after sorting them, so that only those k (pruning_depth) moves are tabulated. This allows for faster calculation, as we don't need to explore every move and choose only those ones that have the best possibility to explore
Pseudocode -

```
For mv ← (all moves for given board, side)
      Evaluate mv using eval
Sort moves in descending order by the evaluated values
Return k best (top k values)
```

Expectinode

Calls minimax on all possible dice moves and returns the sum of min or max
Pseudocode -

```
For dice_rolls ← (all possible dice_rolls)
      Calculate Multiplier * (func_value board,side,dice_roll)
      (where Multiplier is number of dice combinations
             If side explored currently == player_side → func_value=max_func
             Otherwise                                 → func_value=min_func
Return sum of all the calculated values
```

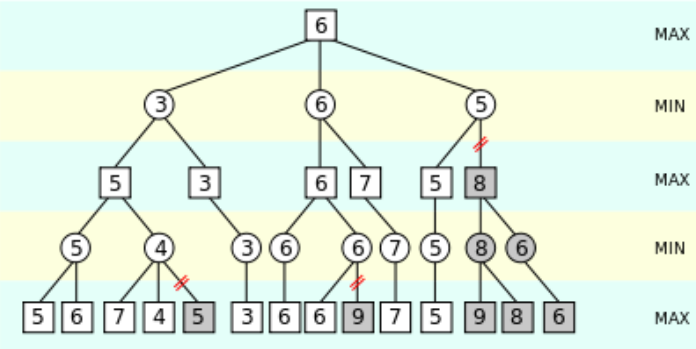
Minimax

Minimize or maximize the evaluated value for the player by calling expectinode with an increase in depth.
If the current turn is the current player's, we use maximize, that calls expectinode with (depth-1) and (opposite side). Expectinode will be minimized, since we want to minimize the value of the opponent's next move (which the AI calculates based on our current move player's

current move). Minimize calls expectinode again with (depth-1) (opposite side - now the same as players), which calls maximize.
Thus, we get a cycle of minimize and maximize, going through expectinode each time - called minimax. The algorithm cycles through till 0 depth is reached.
Pseudocode

```
For mv ← (forwardPruning (all legal moves for board,side,dice))
      newBoard ← perform the move
      Opponent ← opposite side
      Value    ← Expectinode newBoard,opponent,(depth-1)
Return minimum of all values
```

Alpha Beta Pruning
The algorithm performs alpha beta pruning in minimax.



Example of Alpha Beta Pruning. Source: Wikipedia.org
Alpha beta pruning is an adversarial search algorithm and seeks to decrease the number of nodes evaluated by minimax algorithm. The idea behind it is to keep track of alpha and beta values that the minimizing and maximizing algorithm are sure of, that any value crossing those bounds would definitely mean that the move is selected.
Initially, alpha is negative infinity and beta is positive infinity, i.e. the worst possible values. Whenever the maximum score that the minimizing player (i.e. the "beta" player) is assured of becomes less than the minimum score that the maximizing player (i.e., the "alpha" player) is assured of (i.e. beta < alpha), the maximizing player need not consider further descendants of this node, as they will never be reached in the actual play[1].

Best Move

The best move function starts off the calculation for best moves by calling expectiminimax.
Pseudocode:

```
For mv ← (forwardPruning (all legal moves for board,side,dice))
      Value ← Expectinode board,side,dice_roll,depth
Return best move in (sorted values,moves in descending order)
```

[1] "Alpha-Beta Pruning." Wikipedia: The Free Encyclopedia. Wikimedia Foundation, Inc. 18 Dec. 2020. Web. 22 Dec. 2020, en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning

Parallel Solution

Expectinode

We first parallelized expectinode, since that was a direct application of parallelism in this program

Sequential Version

```
sumAllDice func = sum [(multiplier diceRoll)*(func board side currSide
diceRoll alpha beta depth pruningDepth) | diceRoll <- allDiceRolls]
```

Parallel Version

```
sumAllDice func = map
(\diceRoll -> (multiplier diceRoll) *
           func board side currSide diceRoll alpha beta depth pruningDepth)
allDiceRolls `using` parList rdeepseq
```

Here, func is either minimize or maximize based on the current side and the player side
We used rdeepseq since the values have to be evaluated till the end

Expectinode + BestMove + Minimax - Alpha-Beta Pruning

We also removed alpha-beta pruning and parallelized bestMove and minimax algorithms, to see if we improved performance that way as well. We had to remove pruning because it interferes with parallelization, as values are updated every time. Removing it doesn't affect the output, as it only means exploring more nodes.

BestMove Sequential Version

```
bestMove' [] _ bestMoveA = bestMoveA
bestMove' (mv:mvs) bestScore bestMoveA = case (performMoves board side mv)
of ....
```

Where performMoves performs the move on the board, returns a new board and the algorithm then calculates the value for the move using

```
expectiRes = expectinode newBoard side (opposite side) ....
```

BestMove Parallel Version

```
bestMovePar' mvs = snd $ head $ sortBy (\x y -> compare (fst x) (fst y))
(bestMoveParAll mvs)
bestMoveParAll mvs = map innerFunc mvs `using` parList rdeepseq
```

Here, innerFunc performs the moves and calls expectinode on the newBoard

Maximize Sequential Version (Minimize is the same, except minimum value is calculated)

```
minValue' [] _ _ bestScore = bestScore
minValue' (mv:mvs) al bt bestScore = case (performMoves board currSide mv)
of ...
```

Minimize/Maximize and bestMove algorithms are similar, except they return value and move respectively
Here as well, performMoves performs the move on the board, returns a new board and the algorithm then calculates the value for the move using

```
expectiRes = expectinode newBoard side (opposite currSide) (depth-1)
```

Maximize Parallel Version

```
minValuePar' mvs = fst $ head $ sortBy (\x y -> compare (fst x) (fst y))
(minValueParAll mvs)
minValueParAll mvs = map innerFunc mvs `using` parList rdeepseq
```

Here, innerFunc performs the moves and calls expectinode on the newBoard

Evaluation

Evaluation Strategy

We ran the three different versions of the algorithm (sequential, parallel and parallel without Alpha-Beta Pruning) for

- Depths 1 and 2 (any higher and the algorithm takes way too long to run)
- For each depth, we ran it for 5 pruning depths 1..5
- For each (depth,pruning_depth) combination, we ran the parallel version over 2..4 cores (1 would be the same as sequential)
- For each (depth, pruning_depth, core) combination, we ran it 10 times with different seeds (seed controls randomness in the program which means dice roll and choosing of random moves as well for the opponent).
  We averaged the results for each of the 10 seed runs, for each depth, pruning_depth, core combination

Results

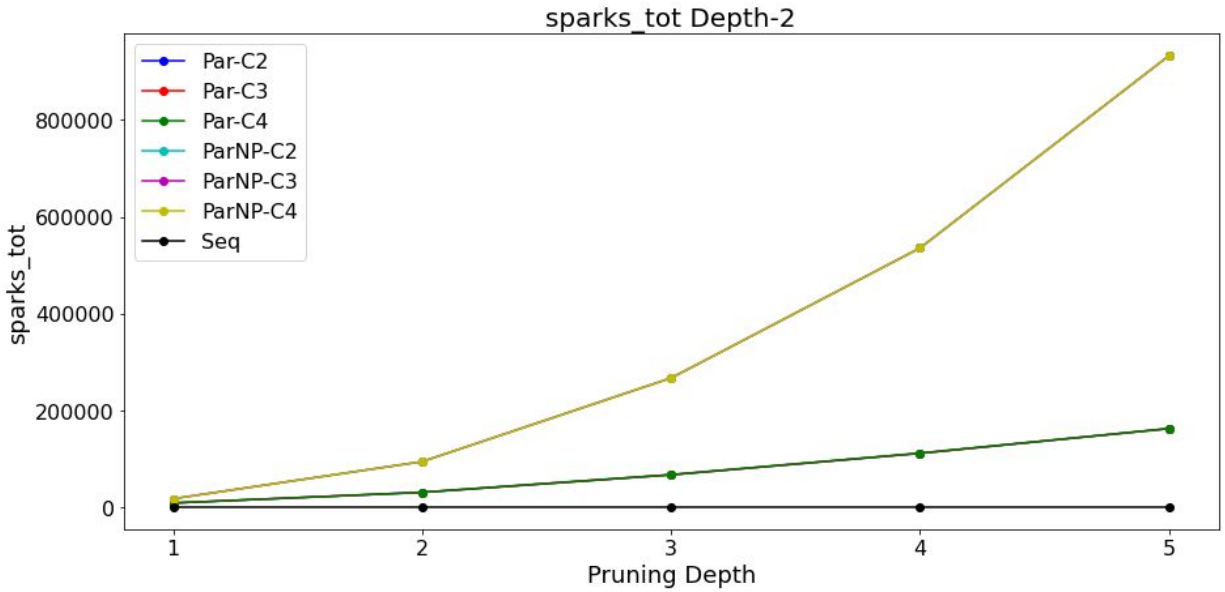
We present results for depth=2 only, as depth=1 is too shallow and doesn't give consistent results.
We first present our data, and then explore the results through graphs to gain a better understanding of them.

Results - Depth=2

prog_type	core	pruning_depth	Time	Total Sp	Converted Sp	GC'D Sp	Fizzled Sp
Par	2	1	1.5446	8870.4	636	1384.8	6849.6
		2	6.701	30756.6	11015.3	2709.8	17031.5
		3	14.3257	67015.2	22836.5	4677.2	39501.5
		4	24.8077	111415.5	31846.9	7085.6	72483
		5	39.9636	162399.3	39623.8	8639	114136.5
	3	1	1.4652	8870.4	1550.9	1230.5	6089
		2	6.3282	30901.5	12346.8	2397.8	16156.9
		3	13.5483	67174.8	26333	3650.6	37191.2
		4	22.8422	111558.3	40514.9	5932.9	65110.5
		5	38.2837	162527.4	54162.4	6945.8	101419.2
	4	1	1.4499	8870.4	2613.7	1088.6	5168.1
		2	6.083	31128.3	13635.4	2609.8	14883.1
		3	12.6969	67526.2	29879.4	4109.3	33537.5
		4	21.6451	111881.7	47041.3	6379.8	58460.6
		5	35.7576	162760.5	64856.9	7584.7	90318.9
ParNP	2	1	1.5178	17760	627.8	8546.5	8585.7
		2	5.864	94363.6	345.4	61825.2	32193
		3	13.8561	267083.5	613.6	194971.7	71498.2
		4	24.7526	535483.3	505	415814.9	119163.4
		5	46.7718	932551.7	583.9	749703.6	182264.2
	3	1	1.4625	17760	1565.5	7752.5	8442
		2	5.597	94364.2	1951.4	58374.6	34038.2
		3	12.9771	267083.6	1402.9	191290.7	74390
		4	23.2512	535483.7	2016.5	407571.1	125896.1
		5	44.7322	932551.8	2235	738500.2	191816.6
	4	1	1.4539	17760.1	2029.6	6895.3	8835.2
		2	5.3892	94363.7	3763.6	55136.1	35464
		3	12.3983	267083.9	3935.9	186778.2	76369.8
		4	22.0413	535512.2	3249.4	407308.5	124954.3
		5	43.2612	932552.1	4590.3	735479.2	192482.6
Seq		1	3.6135	0	0	0	0
		2	15.8631	0	0	0	0
		3	34.1666	0	0	0	0
		4	58.0969	0	0	0	0
		5	95.8851	0	0	0	0

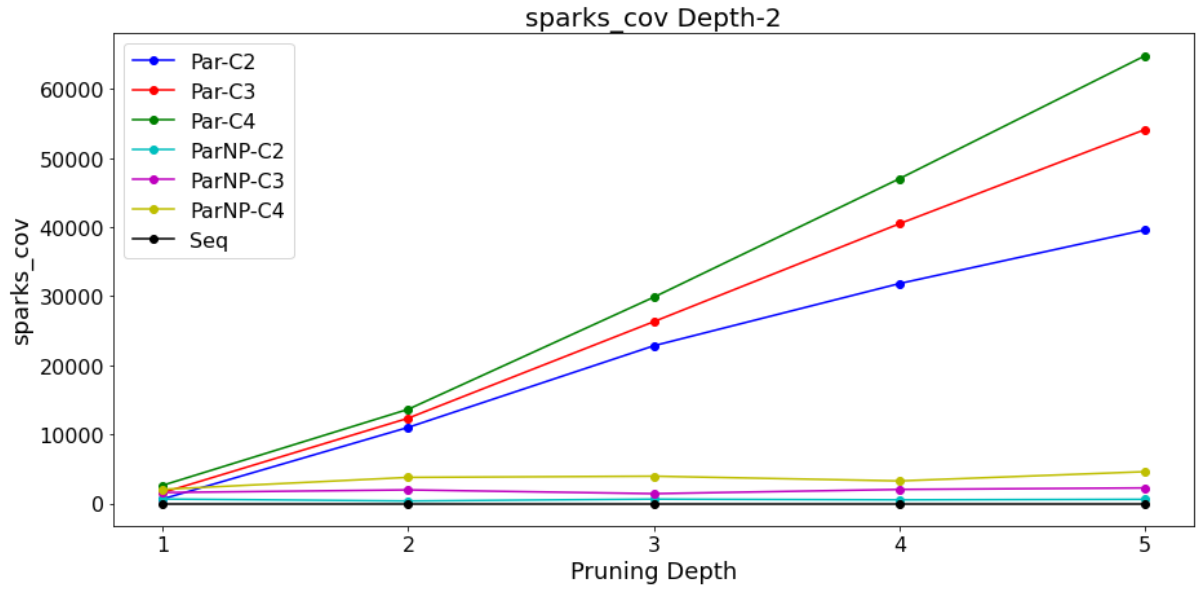
Analysis

Total Sparks



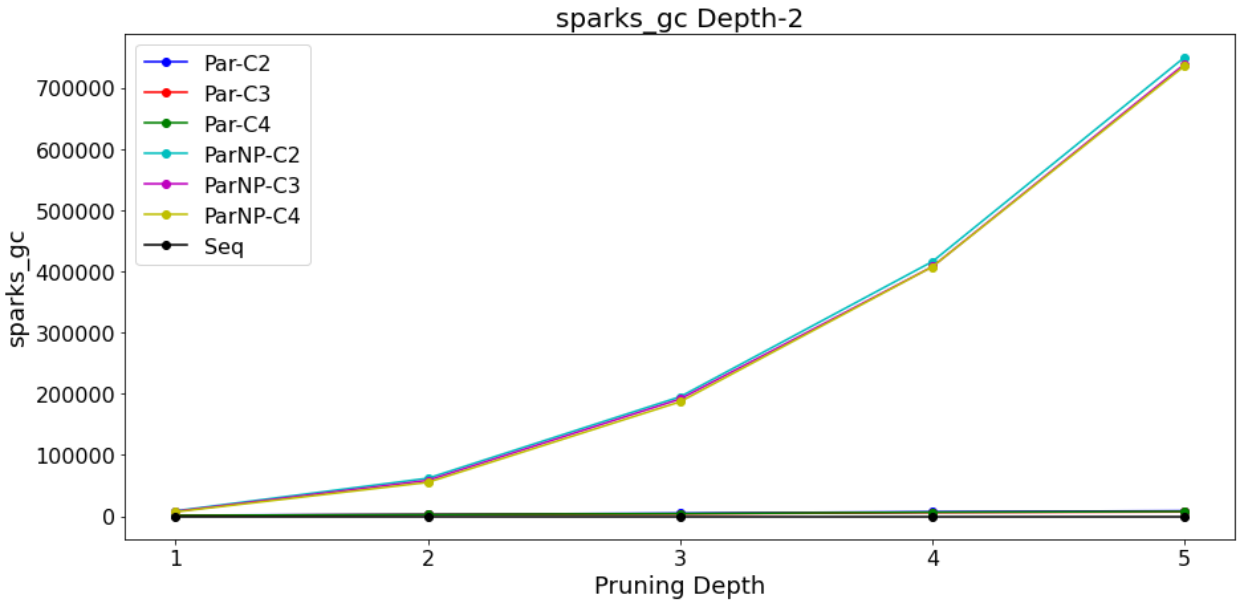
The number of sparks are most for the parallel version without alpha beta pruning, since it evaluates every node till its full depth. It also increases with an increase in pruning depth, as higher depth means less nodes pruned.

Converted Sparks



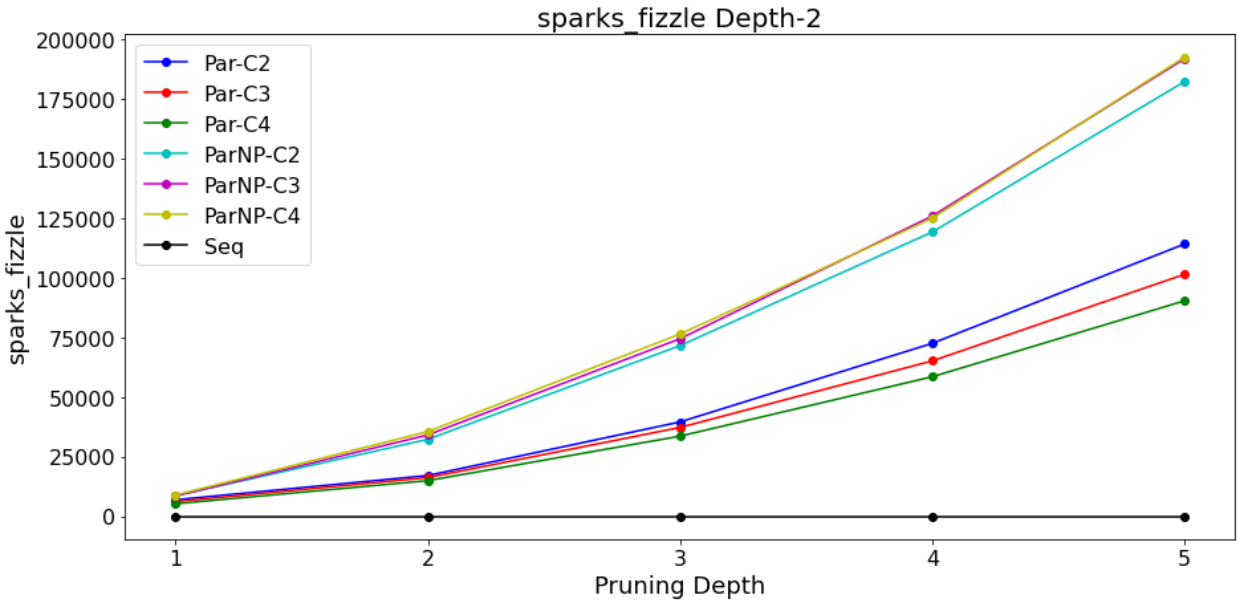
Number of converted sparks increases with increase in pruning depth. It is also the best for the parallel version with alpha-beta pruning, increasing with more core, as the version without pruning has a lot more nodes that fizzle out in forward pruning

GC Sparks



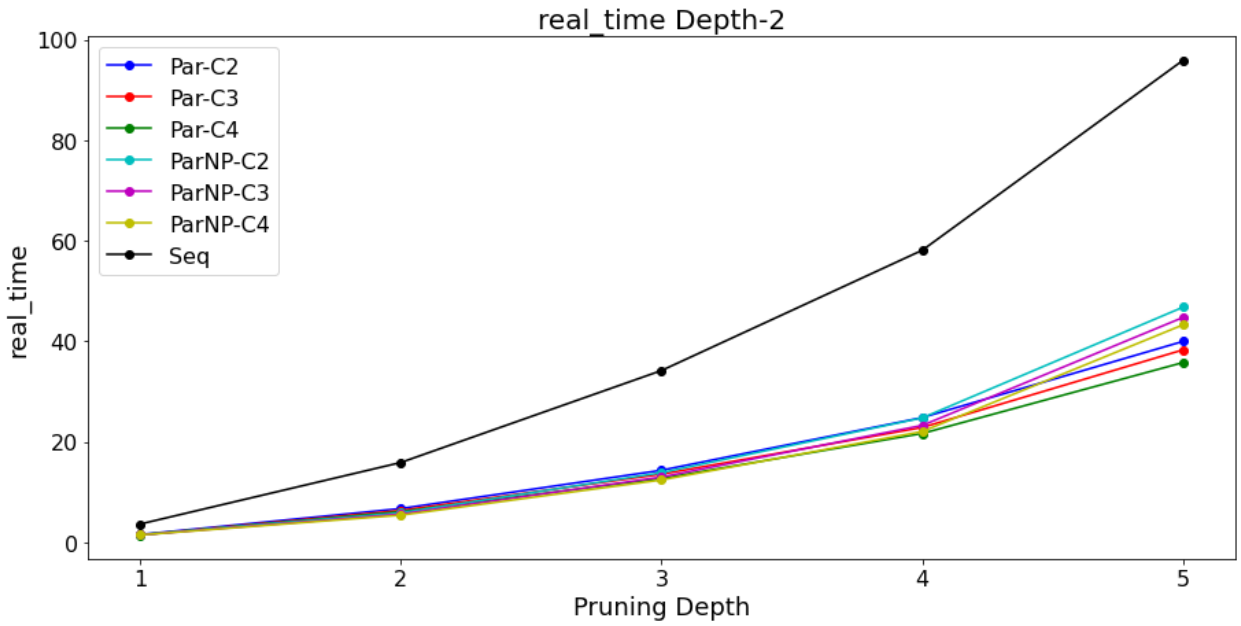
GC is almost the same for all parallel strategies

Fizzle Spark



The number of sparks fizzled increases with pruning depth, which makes sense as more nodes are kept in forward pruning. Sparks fizzled are also higher in the version without alpha-beta pruning, because there are many nodes that are made, however, not evaluated due to forward pruning.

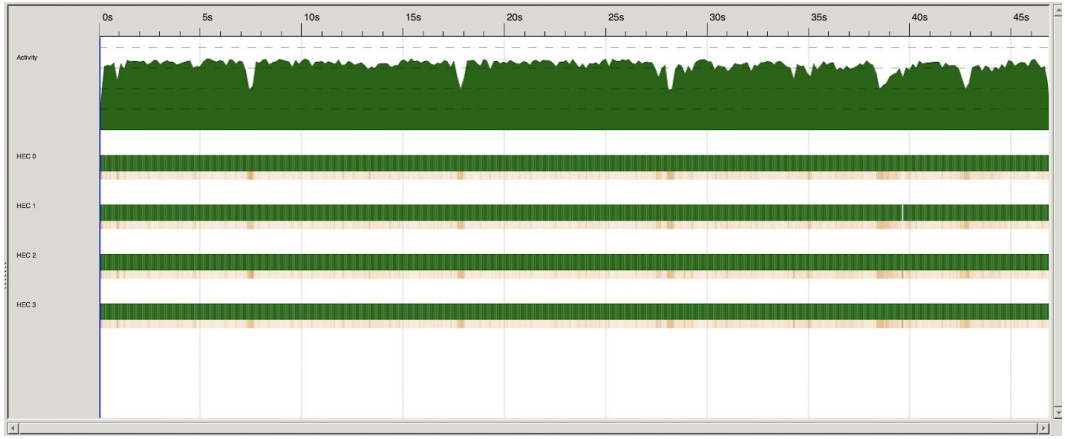
Run time



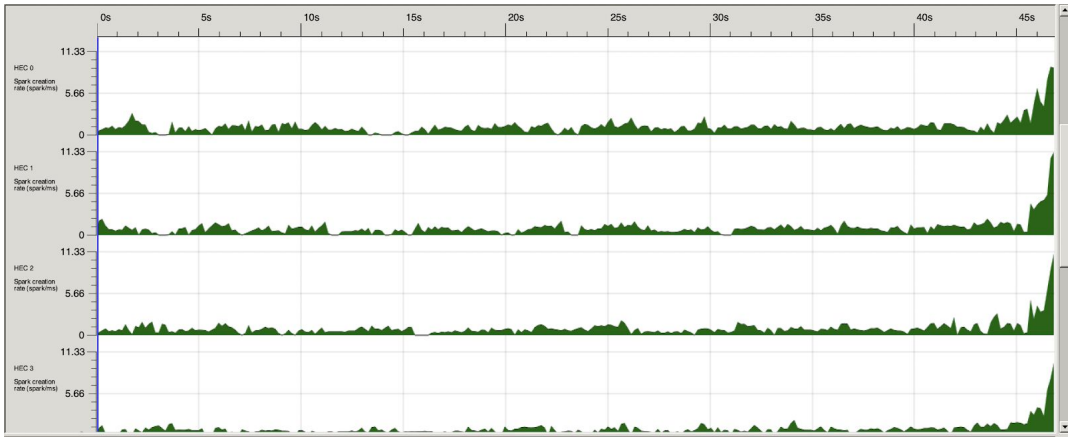
This graph shows a clear distinction between sequential and parallel versions of the program and we get an increase of almost double while using the sequential version. There is not much difference between the pruning and non-pruning version, however, we start to see the difference with an increase in the pruning depth. I think if we continue to increase the pruning depth, or not do forward pruning at all, we would have a substantial increase in run time.

Threadscope

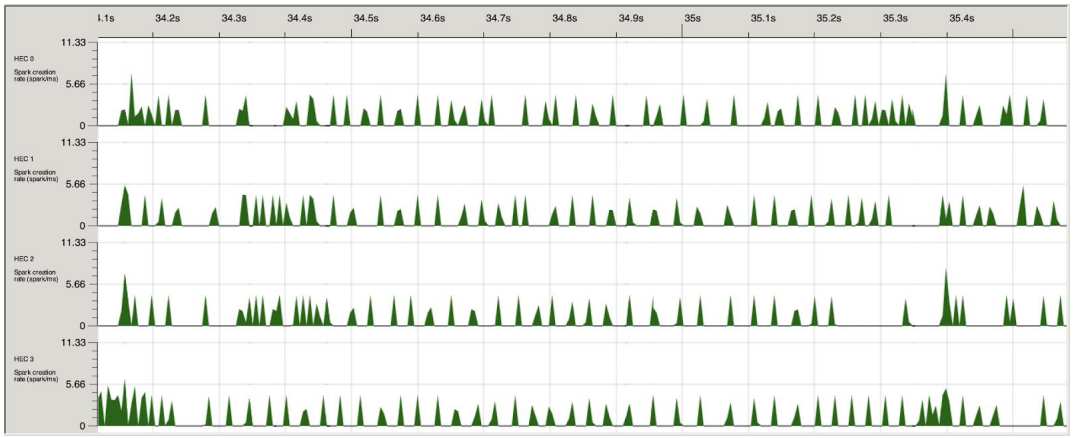
Parallel Version



All cores are almost equally used.

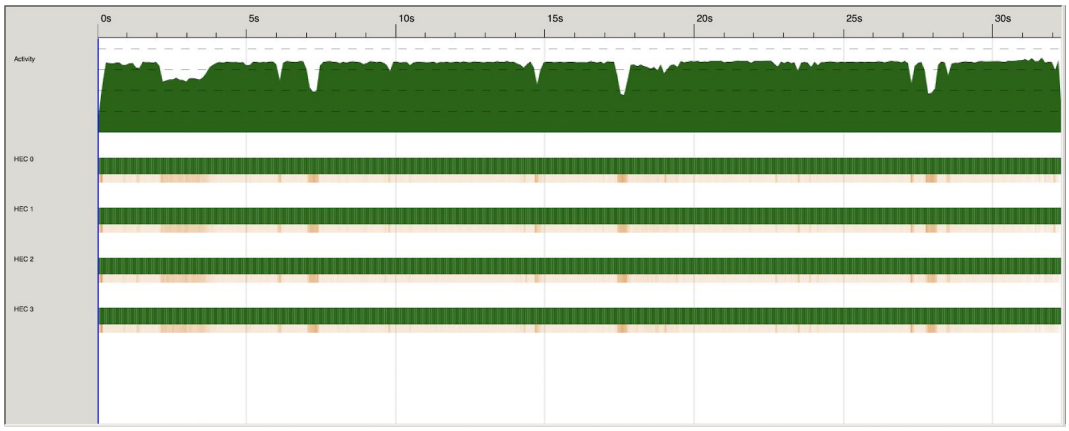


Spark creation is also the same throughout, except the end since that is where the program terminates, and hence almost all nodes come to an end. Looking at spark creation closely:

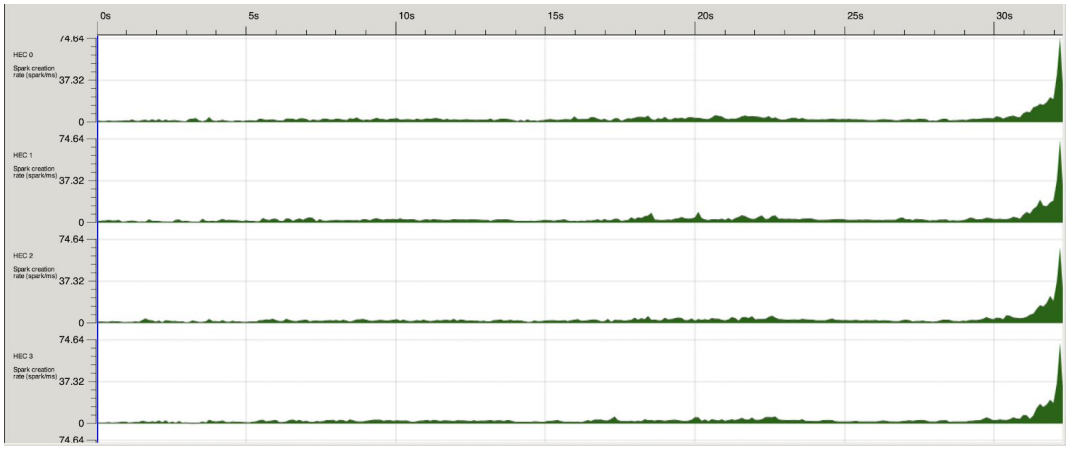


Sparks are created sparsely, since there is a lot of sequential code that is to be done, between each game/within a game as well

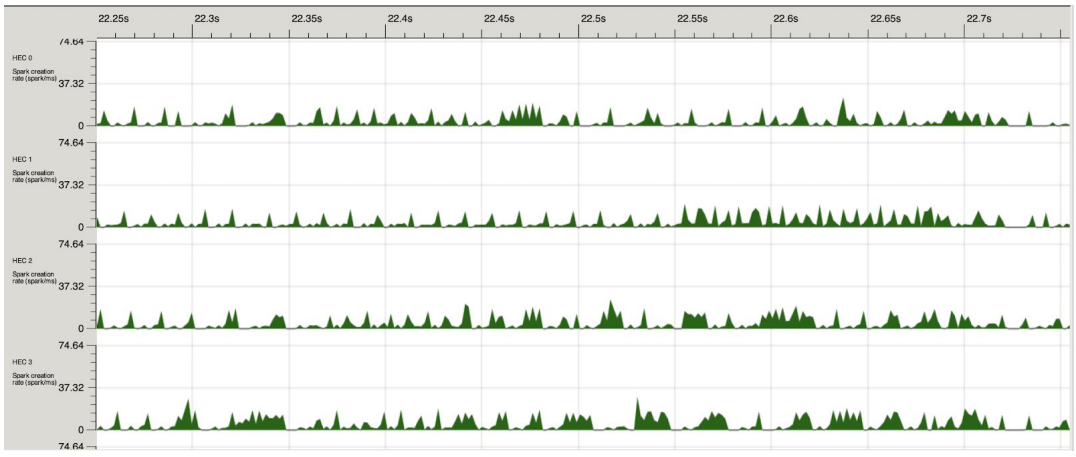
Parallel NP Version (No Alpha-Beta Pruning)



All cores are almost equally used here as well



There is a high amount of spark creation at the end, which overshadows the rest of the graph. Looking at it closely



The sparks created are less sparse, since sparks are now also created instead of sequential alpha-beta pruning.

Code Listing

We have four files

- Backgammon.hs - Helper module
- BackSeq.hs - Module containing sequential implementation of functions
- BackPar.hs - Module containing parallel implementation
- BackParNP.hs - Module containing parallel, No Pruning implementation

The first module is just a helper module. To compile the modules:

- BackSeq.hs - `stack ghc -- BackSeq.hs -threaded -rtsopts -eventlog`
- BackPar.hs - `stack ghc -- BackPar.hs -threaded -rtsopts -eventlog`
- BackParNP.hs - `stack ghc -- BackParNP.hs -threaded -rtsopts -eventlog`

To run the modules, pass the depth, pruningDepth and seed (can be any integer):

- BackSeq.hs - `./BackSeq Black $depth $pruningD $seed +RTS -N4 -s -ls`
- BackPar.hs - `./BackPar Black $depth $pruningD $seed +RTS -N4 -s -ls`
- BackParNP.hs - `./BackParNP Black $depth $pruningD $seed +RTS -N4 -s -ls`

For example:

- `./BackSeq Black 1 2 10 +RTS -N4 -s -ls`
- `./BackPar Black 2 1 4 +RTS -N4 -s -ls`
- `./BackParNP Black 2 3 7 +RTS -N4 -s -ls`

Backgammon.hs

Helper module, contains all functions for gameplay

```
{-
Helper module, that contains all functions required
for the gamePlay
-}

{-# LANGUAGE DeriveGeneric, DeriveAnyClass #-}
module Backgammon (
  Die, Dice, Side(Black, White), Point, Move, Board, Game,
  GameAction, PlayerDecision, GameState,
  InvalidAction, InvalidDecisionType,
  initialBoard, boardTest1, bearOffBoard, barBoard,
  opposite, getChip, allDiceRolls,
  move, legalMoves, performAction, performMoves,
  gamePlay, eval, forwardPruning
)
where
-- import Debug.Trace(trace)
import Control.Monad (foldM)
import qualified Data.Set as Set
import qualified System.Random as R (mkStdGen, randomRs)
import Data.List(sortBy)
import Control.DeepSeq(NFData)
import GHC.Generics (Generic)

type Die = Int
type Dice = (Die, Die)

-- Number of pieces in a triangle
type Chip = Int
data Side = White | Black
  deriving (Eq, Show, Read)
-- Triangle with number and type of pieces
type Point = Maybe (Side, Chip)
type Points = [Point]

type Pos = Int
-- Move from to
data Move = Move Pos Pos
```

```
          | Enter Pos Pos
          | BearOff Pos Pos
  deriving (Eq, Show, Ord, Generic, NFData)
-- type Moves = [Move]

-- Either first dice throw to determine who starts
-- Or an action by player
data GameAction = PlayerAction Side PlayerDecision
                | InitialThrows Die Die
  deriving (Eq, Show)

-- Two play actions possible Move or Throw
data PlayerDecision = Moves [Move]
                    | Throw Dice
  deriving (Eq, Show)

-- Initial throw, 2 player decitions and game end
data GameState = PlayersToThrowInitial
               | ToMove Side Dice
               | ToThrow Side
               | GameFinished Side
  deriving (Eq, Show)

-- Error checks
data InvalidDecisionType = NoPieces Pos
                         | MovedOntoOpponentsClosedPoint Pos
                         | NoBarPieces Side
  deriving (Eq, Show)

-- Error checks
data InvalidAction = ActionInvalidForState GameState GameAction
                   | InvalidPlayerDecision Game PlayerDecision
InvalidDecisionType
  deriving (Eq, Show)

-- triangles and chips, barWhite, barBlack
data Board
  = Board Points Int Int
  deriving (Eq, Show)

-- Storing game at every turn
data Game = Game { gameBoard :: Board,
                   gameActions :: [GameAction],
```

```
                  gameState :: GameState}
  deriving (Eq)

instance Show Game where
  show game = "Game Board: " ++ show board ++
              "\n\nGame Actions: " ++ show actions ++
              "\n\nGame State: " ++ show state where
    board = gameBoard game
    actions = if (length gActions >=6)
              then (show $ take 3 gActions) ++ "........." ++ (show $
takeLast 3 gActions)
              else show gActions
    state = gameState game
    gActions = gameActions game

-- splitComm xs = split xs ','
--
-- split :: String -> Char -> [String]
-- split [] delim = [""]
-- split (c:cs) delim
--     | c == delim = "" : rest
--     | otherwise = (c : head rest) : tail rest
--     where
--         rest = split cs delim

-- Start with this board
initialBoard :: Board
initialBoard = Board [ Nothing, Just (White, 2), Nothing, Nothing, Nothing,
Nothing, Just (Black, 5), Nothing, Just (Black, 3), Nothing, Nothing,
Nothing, Just (White, 5),
                       Just (Black, 5), Nothing, Nothing, Nothing, Just
(White, 3), Nothing, Just (White, 5), Nothing, Nothing, Nothing, Nothing,
Just (Black, 2), Nothing
                     ] 0 0

-- Can try bearing off with this board
-- Bear off Black 1 dice roll,  Black 2 dice roll,  Black 1,2 dice rolls
bearOffBoard :: Board
bearOffBoard = Board [ Nothing, Nothing, Just (Black, 3), Nothing, Nothing,
Nothing, Nothing, Nothing, Nothing, Nothing, Nothing, Nothing, Nothing,
                       Nothing, Nothing, Nothing, Nothing, Nothing, Nothing,
Nothing, Nothing, Nothing, Nothing, Nothing, Just (White, 2), Nothing
                     ] 0 0
```

```haskell
-- Board with black chips on bar
-- Call with dieroll 2
barBoard :: Board
barBoard = Board [ Nothing, Nothing, Just (Black, 3), Just (White, 3),
Nothing, Nothing, Nothing, Nothing, Nothing, Nothing, Nothing, Nothing,
Nothing,
                   Nothing, Nothing, Nothing, Nothing, Nothing, Nothing,
Nothing, Nothing, Nothing, Nothing, Nothing, Nothing, Nothing
                   ] 0 2

-- Random Test Board
boardTest1 :: Board
boardTest1 = Board [ Nothing, Just (Black, 2), Just (Black, 5), Just
(White, 1), Just (Black, 2), Nothing, Nothing, Nothing, Nothing, Nothing,
Nothing, Nothing, Nothing, Nothing,
                     Nothing, Nothing, Nothing, Nothing, Nothing, Nothing,
Just (White, 5), Nothing, Just (White, 2), Just (White, 5), Just (Black,
1), Just (White, 2), Nothing
                     ] 0 0

-- Test Board with elements in endzone
endBoard :: Board
endBoard = Board [ Nothing, Just (White,1),  Nothing, Nothing, Nothing,
Nothing, Just (Black, 5), Nothing, Just (Black, 3), Nothing, Nothing,
Nothing, Just (White, 5),
                   Just (Black, 5), Nothing, Nothing, Nothing, Just (White,
3), Nothing, Just (White, 5), Nothing, Nothing, Nothing, Nothing, Just
(Black, 2), Nothing
                   ] 0 0


allDiceRolls :: [Dice]
allDiceRolls = [(i,j) | i <- [1..6], j <- [i..6]]


--------------- Helper Functions ---------------
-- Take the last n elements from xs
takeLast :: Int -> [a] -> [a]
takeLast n xs = drop (length xs - n) xs

newGame :: Game
newGame = Game initialBoard [] PlayersToThrowInitial
```

```haskell
-- Black goes from 24 -> 1, white from 1 -> 24
direction :: Side -> Int
direction White = 1
direction Black = -1

-- Opposite sides
opposite :: Side -> Side
opposite White = Black
opposite Black = White

-- Get the chip at position from Board
getChip :: Board -> Pos -> Point
getChip (Board b _ _) pos = b !! (pos)

-- Returns False if there are any chips on any triangle for the given side
-- In the given points
checkChipSide :: [Point] -> Side -> Bool
checkChipSide [] _ = True
checkChipSide (p:pts) side = case p of
  -- If Nothing, then move on to next point
  Nothing -> checkChipSide pts side
  -- Otherwise, check if chips side == given side
  Just (s,_)  | s==side -> False
              | otherwise -> checkChipSide pts side

dieList :: Dice -> [Die]
dieList (d1, d2) =
  if d1 == d2 then [d1, d1, d1, d1]
              else [d1, d2]

-- increase bar value
incBar :: Side -> Board -> Board
incBar White (Board b bw bb) = Board b (bw+1) bb
incBar Black (Board b bw bb) = Board b bw     (bb+1)

-- decrease bar value
decBar :: Side -> Board -> Either InvalidDecisionType Board
decBar side (Board b bw bb) | side==White && bw>0 = Right (Board b (bw-1)
bb)
                            | side==Black && bb>0 = Right (Board b bw
(bb-1))
                            | otherwise = Left (NoBarPieces White)
```

```haskell
--------------- Functions ---------------
-- move chip from 'from' to 'to'
-- bear off handled by moving and removing the piece from the board
-- Enter (bar) handled by subtracting from bar
move :: Side -> Board -> Move -> Either InvalidDecisionType Board
move side board (Move from to) = handleMoves board side from to
move side board (BearOff from to) = handleBearOffMove (handleMoves board
side from to) side
move side board (Enter from to) = takePiece from board side >>= landPiece
to side

-- Handle regular moves
handleMoves :: Board -> Side -> Pos -> Pos -> Either InvalidDecisionType
Board
handleMoves board side from to =
  case getChip board from of
    Just (_,_)  -> takePiece from board side >>= landPiece to side
    Nothing     -> Left (NoPieces from)

-- Handle Bear off moves
handleBearOffMove :: Either InvalidDecisionType Board -> Side -> Either
InvalidDecisionType Board
handleBearOffMove board side =
  case board of
    Right (Board bd bw bb)  | side==Black -> Right (Board ([Nothing] ++
(tail bd)) bw bb)
                            | side==White -> Right (Board ((take 25 bd) ++
[Nothing]) bw bb)
    Right (Board _ _ _)     -> board
    Left err                -> Left err

-- take piece from the board, throws error on nonlegality
takePiece :: Pos -> Board -> Side -> Either InvalidDecisionType Board
takePiece (-1) board side = decBar side board
takePiece pos board@(Board b bw bb) _ =
  case getChip board pos of
    Just (s, n) -> Right (Board (take (pos) b ++ [dec1 s n] ++ drop (pos+1)
b) bw bb)
    Nothing     -> Left (NoPieces pos)
  where
```

```
    dec1 _ 1 = Nothing
    dec1 s n = Just (s, n-1)

-- add piece to location, throws error on nonlegality
landPiece :: Pos -> Side -> Board -> Either InvalidDecisionType Board
landPiece pos side board@(Board b bw bb) =
  case getChip board pos of
    Nothing                 -> Right (setField (Just (side, 1)))
    Just (s, n) | s == side -> Right (setField (Just (side, n+1)))
    Just (_, n) | n == 1    -> Right (incBar (opposite side) (setField
(Just (side, 1))))
    _                       -> Left (MovedOntoOpponentsClosedPoint pos)
  where
    setField f = Board (take (pos) b ++ [f] ++ drop (pos+1) b) bw bb

-- Get legal moves for a single dice (handles any value 1..36)
-- get_normal_moves from backgammon.py
singleDieLegalMoves :: Board -> Die -> Side -> [Move]
singleDieLegalMoves bd d side = moves 1 where
  moves :: Pos -> [Move]
  moves 25 = []
  moves i2 =
    case getChip bd i2 of
      Nothing -> nextMoves
      -- if side same as chip, and 1 <= pos,move_pos <= 24 (in the board)
      Just (s,_) -> if (s == side && i2<=24 && ni <= 24 && i2>=1 && ni>=1)
                    then case getChip bd ni of
                      -- Check if move_pos is legal
                      Nothing -> Move i2 ni : nextMoves
                      Just (s2,n2)  | s2==side ->   Move i2 ni : nextMoves
                                    | otherwise ->  if (n2==1)
                                                    then Move i2 ni :
nextMoves
                                                    else nextMoves
                    else nextMoves
      where nextMoves = moves (i2+1)
            -- find next move_pos
            ni = (i2 + d * direction side)


-- Checks if bearing off is possible
-- Only possible if
-- For Black = no chips on any triangle b/w [7..24]
```

```
-- For White = no chips on any triangle b/w [1..18]
canBearOff :: Board -> Side -> Bool
canBearOff (Board b bw bb) side
  | side==Black = (bb==0) && checkChipSide (takeLast 19 b) side
  | otherwise   = (bw==0) && checkChipSide (take 19 b) side

-- play bear off move, assumes bear off possible
bearOffMoves :: Board -> Die -> Side -> [Move]
bearOffMoves bd dieRoll side =
  directMoves ++ homeMoves ++ bigBearOff where
    -- direct bearoffs (if chip at 5 away from bearoff and die roll 5)
    directMoves :: [Move]
    directMoves | not (checkChipSide [(getChip bd ind)] side) = [(BearOff
ind end)]
                | otherwise = [] where
                  ind = if side==White then (25-dieRoll) else dieRoll
    -- Single die moves for dieRoll, within homeboard no bearing off
    homeMoves = singleDieLegalMoves bd dieRoll side
    directHome = directMoves ++ homeMoves
    -- If nth works out, then you can bearOff from indexes < dieRoll
    bigBearOff  | length(directHome)==0 = bigBearOffFunc 1
                | otherwise = [] where
                  bigBearOffFunc 7 = []
                  bigBearOffFunc i =
                    case getChip bd ind2 of
                      Nothing -> bigBearOffFunc (i+1)
                      Just (s,_)  | s==side && i<dieRoll -> BearOff ind2
end : bigBearOffFunc (i+1)
                                  | otherwise -> bigBearOffFunc (i+1)
                      where ind2 = if side==White then (25-i) else i
    end = if side==White then 25 else 0


barMoves :: Board -> Die -> Side -> [Move]
barMoves bd dieRoll side =
  case getChip bd ind of
    Nothing -> [(Enter (-1) ind)]
    Just (s,_)  | s==side -> [(Enter (-1) ind)]
                | otherwise -> []
    where ind = if side==White then (25-dieRoll) else dieRoll

-- Keep only unique moves from list of moves.
-- Moves can be reversed as well
```

```haskell
uniqueMoves :: [[Move]] -> [[Move]]
uniqueMoves xs = uniqueMoves' Set.empty xs where
  uniqueMoves' :: Set.Set([Move]) -> [[Move]] -> [[Move]]
  uniqueMoves' _ [] = []
  uniqueMoves' s (x:xss)
   | x `Set.member` s || (reverse x) `Set.member` s = uniqueMoves' s xss
   | otherwise = x : uniqueMoves' (Set.insert x s) xss

-- Given a dice roll, board and a side, it gives legal moves
-- Moves can be run on function move directly
-- Checks bear offs, bar and normal moves as well
-- Handles double dice rolls and permutations of dice
legalMoves :: Board -> Dice -> Side -> [[Move]]
legalMoves bdM dice side
  -- if both die values same, double dice roll
  | (length dieRollsM == 4) = legalMoves' (Right bdM) dieRollsM
  -- otherwise dice + reverse dice
  | otherwise = uniqueMoves (legalMoves' (Right bdM) dieRollsM ++
                             legalMoves' (Right bdM) (reverse dieRollsM))
  where dieRollsM = dieList dice
        legalMoves' :: Either InvalidDecisionType Board -> [Die] ->
[[Move]]
        legalMoves' _ [] = [[]]
        legalMoves' (Left _) _ = [[]]
        legalMoves' (Right bd@(Board _ bw bb)) dieRolls
          -- check bar moves
          | (side==White && bw/=0) || (side==Black && bb/=0) =
            if (length bMoves /= 0)
            then [m:ms |  m <- bMoves,
                          ms <- legalMoves' (move side bd m) nDRolls]
            else legalMoves' (Right bd) nDRolls
          -- check bear off moves
          -- check single die move
          | (length nMoves /= 0) =
            [m:ms | m <- nMoves,
                    ms <- legalMoves' (move side bd m) nDRolls]
          -- if no move possible with die, then go to the next die
          | otherwise = legalMoves' (Right bd) nDRolls where
              dRoll = head dieRolls
              nDRolls = tail dieRolls
              bMoves = barMoves bd dRoll side
              nMoves =  if (canBearOff bd side)
                        then (bearOffMoves bd dRoll side)
```

```
                    else (singleDieLegalMoves bd dRoll side)

-- change game state to given state
moveToState :: GameState -> Game -> Game
moveToState state game = game { gameState = state }

-- add new action to action list in game
appendAction :: GameAction -> Game -> Game
appendAction action game = game { gameActions = gameActions game ++
[action] }

-- change state and add action for a game
success :: Game -> GameState -> GameAction -> Either InvalidAction Game
success game state action =
  Right ((appendAction action . moveToState state) game)

-- stop at the first error, otherwise continue
first :: (a -> c) -> Either a b -> Either c b
first f (Left l)  = Left (f l)
first _ (Right r) = Right r

-- check if game has ended (no pieces left on board of side)
checkGameEnd :: Board -> Side -> Bool
checkGameEnd (Board b bw bb) side = (bar==0) && (checkChipSide b side)
where
  bar = if side==White then bw else bb

-- handles the different actions
performAction :: GameAction -> Game -> Either InvalidAction Game
-- Initial dice throw
performAction act@(InitialThrows dw db) game@Game{gameState =
PlayersToThrowInitial} =
  success game (if dw /= db then ToMove side (normDice (dw,db))
                else PlayersToThrowInitial) act where
                  side = if dw > db then White else Black
-- Move
performAction act@(PlayerAction pSide m@(Moves moves)) game@Game{gameState
= ToMove side _} | pSide==side =
  do  updatedBoard <- wrapInInvalidDecision (foldM (move side) board moves)
      if (checkGameEnd updatedBoard side)
      then success (game {gameBoard = updatedBoard}) (GameFinished side)
act -- Game finished
      else success (game {gameBoard = updatedBoard}) (ToThrow (opposite
```

```
side)) act where -- If not finished, then throw for opposite side
            board = gameBoard game
            wrapInInvalidDecision = first (InvalidPlayerDecision game m)
-- Throw
performAction act@(PlayerAction pSide (Throw dice)) game@Game{gameState =
ToThrow side} | pSide==side =
  success game (ToMove side dice) act
-- Any other action is invalid
performAction action game = Left (ActionInvalidForState (gameState game)
action)

-- Max die first
normDice :: Dice -> Dice
normDice (d1, d2) = if d1 > d2 then (d1, d2) else (d2, d1)

-- White starts
initialThrowWhite :: Int -> GameAction
initialThrowWhite seed = head [ makeInitialAction (normDice (i,j)) | (i,j)
<- nRolls seed 10, i/=j] where
                               makeInitialAction (a,b)= InitialThrows a b

-- Random starts
initialThrowRandom :: Int -> GameAction
initialThrowRandom seed = head [ makeInitialAction (i,j) | (i,j) <- nRolls
seed 10, i/=j] where
                                 makeInitialAction (a,b)= InitialThrows a
b

-- Get n dice rolls
nRolls :: Int -> Int -> [Dice]
nRolls seed n = zip (take n s1) (take n s2)
                 where s1 = R.randomRs (1,6) (R.mkStdGen seed) :: [Die]
                       s2 = R.randomRs (1,6) (R.mkStdGen (seed+1)) ::
[Die]

-- wrapper for 1000 dice rolls
diceRolls :: Int -> [Dice]
diceRolls seed = nRolls seed 1000

getRandomMove :: [[Move]] -> Int -> [Move]
getRandomMove [] _ = []
getRandomMove moves seed = moves !! (head $ R.randomRs (0,((length
moves)-1)) (R.mkStdGen seed) :: Int)
```

```haskell
-- loops through and plays a game
-- game always starts with white, change initialThrowWhite to
initialThrowRandom to random start
-- handles state and then loop, ends at game end only
-- seed defines randomness, reproducable results
gamePlay :: Side -> p1 -> p2 -> Int -> (Board -> Dice -> Side -> p1 -> p2
-> [Move]) -> Either InvalidAction Game
gamePlay pSide depth pruningDepth seed bestMoveFunc= gamePlay' newGame 1
where
  dRolls = diceRolls seed
  gamePlay' :: Game -> Int -> Either InvalidAction Game
  -- handle initial throw
  gamePlay' game@Game{gameState = PlayersToThrowInitial} n =
    do
      nextGame <- performAction (initialThrowWhite seed) game
      gamePlay' nextGame n
  -- Move
  -- Chooses first move
  gamePlay' game@Game{gameState = ToMove side dice} n =
    do
      let board = gameBoard game
      let validMoves = legalMoves board dice side
      -- let randomMove = if (length validMoves > 0) then (head validMoves)
else []
      let randomMove = getRandomMove validMoves seed
      let moveI = if pSide == side
                  then bestMoveFunc board dice side depth pruningDepth
                  else randomMove
      nextGame <- performAction (PlayerAction side (Moves moveI)) game
      gamePlay' nextGame n
  -- Dice throw
  gamePlay' game@Game{gameState = ToThrow side} n =
    do
      nextGame <- performAction (PlayerAction side (Throw (dRolls !! n)))
game
      gamePlay' nextGame (n+1)
  -- End game
  gamePlay' game@Game{gameState = GameFinished _} _ = Right (game)

-- Helper func - Takes a list of points, and returns a list of Ints
-- +1 for White at every point
-- -1 for Black at every point
```

```haskell
pointCounter :: Point -> Int
pointCounter point = case point of
  Nothing -> 0
  Just(a,b) -> case a of
    White -> b
    Black -> (-b)

-- calculates chips on home board
homeBoardChips :: Board -> Side -> Int
homeBoardChips bd side = sum [(checkChip i) | i <- range] where
  range = if side==White then [19..24] else [1..6]
  checkChip ind = case getChip bd ind of
    Nothing -> 0
    Just (s,n) -> if (s==side) then n else 0

eval :: Board -> Side -> Int
-- eval bd@(Board b bw bb) side = trace (show $ [distance, barWeight,
homeWin, homeChips, opponentChips]) finalValue where
eval bd@(Board b bw bb) side = finalValue where
  boardValues = map pointCounter b
  whitePieces = sum $ filter (>0) boardValues
  blackPieces = abs $ sum $ filter (<0) boardValues
  distanceList = case side of
    White -> filter (>0) $ zipWith (*) [24, 23..1] $ tail boardValues
    Black -> filter (<0) $ zipWith (*) [1..24] $ tail boardValues
  distance = abs $ sum distanceList
  barWeight = case side of
    White -> bw
    Black -> bb
  homeWin = case side of
    White -> 15 - bw - whitePieces
    Black -> 15 - bb - blackPieces
  -- opponentWin = case side of
  --   White -> 15 - bb - blackPieces
  --   Black -> 15 - bw - whitePieces
  homeChips = homeBoardChips bd side
  opponentChips = (homeBoardChips bd (opposite side))
  finalValue = homeChips + 10 * homeWin - distance - 10 * barWeight -
opponentChips

-- Performs the given move
performMoves :: Board -> Side -> [Move] -> Either InvalidDecisionType Board
performMoves board _ [] = (Right board)
```

```haskell
performMoves board side (m:ms) = case (move side board m) of
  (Left l) -> Left l
  (Right newBoard) -> performMoves newBoard side ms

-- forwardPruning Algorithm
forwardPruning :: Board -> Side -> [[Move]] -> Int -> [[Move]]
forwardPruning board side moves k
  | length moves < k = moves
  | otherwise = [mv | (_,mv) <- (take k sortedFordwardPruningList)] where
    sortedFordwardPruningList = sortBy (\x y -> compare (fst x) (fst y))
fordwardPruningList
    fordwardPruningList = zip [-1*(fordwardPruning' mv) | mv <- moves]
moves
    fordwardPruning' mv = case (performMoves board side mv) of
      (Left _) -> 0
      (Right newBoard) -> eval newBoard side
```

BackSeq.hs

```
{-
Sequential implementation of functions, uses backgammon.hs module
Check Project report for explaination of each function.
-}
import Backgammon
import System.Environment(getArgs)

bestMove :: Board -> Dice -> Side -> Int -> Int -> [Move]
bestMove board diceRoll side depth pruningDepth = bestMove'
forwardPruningMoves (-1/0) [] where
  allLegalMoves = legalMoves board diceRoll side
  forwardPruningMoves = forwardPruning board side allLegalMoves
pruningDepth
  bestMove' :: (Ord t, Fractional t) => [[Move]] -> t -> [Move] -> [Move]
  bestMove' [] _ bestMoveA = bestMoveA
  bestMove' (mv:mvs) bestScore bestMoveA = case (performMoves board side
mv) of
    (Left _) -> bestMove' mvs bestScore bestMoveA
    (Right upBoard) -> bestMove' mvs newBestScore newBestMove where
      expectiRes = expectinode upBoard side (opposite side) bestScore (1/0)
depth pruningDepth
      newBestScore = if (expectiRes>bestScore) then expectiRes else
bestScore
      newBestMove = if (expectiRes>bestScore) then mv else bestMoveA


expectinode :: (Ord t, Fractional t) => Board -> Side -> Side -> t -> t ->
Int -> Int -> t
expectinode board side _ _ _ 0 _ = fromIntegral $ eval board side
expectinode board side currSide alpha beta depth pruningDepth
  | side==currSide = sumAllDice minValue
  | otherwise = sumAllDice maxValue where
    sumAllDice func = sum [(multiplier diceRoll)*(func board side currSide
diceRoll alpha beta depth pruningDepth)
                           | diceRoll <- allDiceRolls]
    multiplier (d1,d2) = if (d1==d2) then (1/36) else (1/18)

minValue :: (Fractional t, Ord t) => Board -> Side -> Side -> Dice -> t ->
t -> Int -> Int -> t
```

```
minValue board side currSide diceRoll alpha beta depth pruningDepth
  | length allLegalMoves > 0 = minValue' forwardPruningMoves alpha beta
(1/0)
  | otherwise = expectinode board side (opposite currSide) alpha beta
(depth-1) pruningDepth where
    allLegalMoves = legalMoves board diceRoll currSide
    forwardPruningMoves = forwardPruning board currSide allLegalMoves
pruningDepth
    minValue' :: (Ord t, Fractional t) => [[Move]] -> t -> t -> t -> t
    minValue' [] _ _ bestScore = bestScore
    minValue' (mv:mvs) al bt bestScore = case (performMoves board currSide
mv) of
      (Left _) -> minValue' mvs al bt bestScore
      (Right newBoard) -> if newBestScore <= al
                          then newBestScore
                          else minValue' mvs al newBt newBestScore where
        expectiRes = expectinode newBoard side (opposite currSide) al bt
(depth-1) pruningDepth
        newBestScore = min bestScore expectiRes
        newBt = min bt newBestScore

maxValue :: (Fractional t, Ord t) => Board -> Side -> Side -> Dice -> t ->
t -> Int -> Int -> t
maxValue board side currSide diceRoll alpha beta depth pruningDepth
  | length allLegalMoves > 0 = maxValue' forwardPruningMoves alpha beta
(-1/0)
  | otherwise = expectinode board side (opposite currSide) alpha beta
(depth-1) pruningDepth where
    allLegalMoves = legalMoves board diceRoll currSide
    forwardPruningMoves = forwardPruning board currSide allLegalMoves
pruningDepth
    maxValue' :: (Ord t, Fractional t) => [[Move]] -> t -> t -> t -> t
    maxValue' [] _ _ bestScore = bestScore
    maxValue' (mv:mvs) al bt bestScore = case (performMoves board currSide
mv) of
      (Left _) -> maxValue' mvs al bt bestScore
      (Right newBoard) -> if newBestScore >= bt
                          then newBestScore
                          else maxValue' mvs newAl bt newBestScore where
        expectiRes = expectinode newBoard side (opposite currSide) al bt
(depth-1) pruningDepth
        newBestScore = max bestScore expectiRes
        newAl = max al newBestScore
```

```haskell
main :: IO()
main = do
  args <- getArgs
  let side = read $ head args :: Side
  let restArgs = map (\x -> (read x :: Int)) (tail args)
  let depth = (restArgs !! 0)
  let pruningDepth = (restArgs !! 1)
  let seed = (restArgs !! 2)
  let ans = gamePlay side depth pruningDepth seed bestMove
  print $ ans
```

BackPar.hs

```
{-
Parallel implementation of functions, uses backgammon.hs module
Check Project report for explaination of each function.
-}

import Backgammon
import Control.Parallel.Strategies(using, parList, rdeepseq)
import Control.DeepSeq(NFData)
import System.Environment(getArgs)

bestMovePar :: Board -> Dice -> Side -> Int -> Int -> [Move]
bestMovePar board diceRoll side depth pruningDepth = bestMovePar'
forwardPruningMoves ((-1/0)::Double) [] where
  allLegalMoves = legalMoves board diceRoll side
  forwardPruningMoves = forwardPruning board side allLegalMoves
pruningDepth
  bestMovePar' :: (Ord t, Fractional t, NFData t, Num t) => [[Move]] -> t
-> [Move] -> [Move]
  bestMovePar' [] _ bestMoveA = bestMoveA
  bestMovePar' (mv:mvs) bestScore bestMoveA = case (performMoves board side
mv) of
    (Left _) -> bestMovePar' mvs bestScore bestMoveA
    (Right upBoard) -> bestMovePar' mvs newBestScore newBestMove where
      expectiRes = expectinodePar upBoard side (opposite side) bestScore
(1/0) depth pruningDepth
      newBestScore = if (expectiRes>bestScore) then expectiRes else
bestScore
      newBestMove = if (expectiRes>bestScore) then mv else bestMoveA

expectinodePar :: (Fractional a, NFData a, Ord a) => Board -> Side -> Side
-> a -> a -> Int -> Int -> a
expectinodePar board side _ _ _ 0 _ = fromIntegral $ eval board side
expectinodePar board side currSide alpha beta depth pruningDepth
  | side==currSide = sum $ sumAllDice minValuePar
  | otherwise = sum $ sumAllDice maxValuePar where
    sumAllDice func = map (\diceRoll -> (multiplier diceRoll) *
                                  func board side currSide diceRoll alpha
beta depth pruningDepth) allDiceRolls
                                  `using` parList rdeepseq
    multiplier (d1,d2) = if (d1==d2) then (1/36) else (1/18)
```

```
minValuePar :: (Fractional t, Ord t, NFData t) => Board -> Side -> Side ->
Dice -> t -> t -> Int -> Int -> t
minValuePar board side currSide diceRoll alpha beta depth pruningDepth
  | length allLegalMoves > 0 = minValuePar' forwardPruningMoves alpha beta
(1/0)
  | otherwise = expectinodePar board side (opposite currSide) alpha beta
(depth-1) pruningDepth where
    allLegalMoves = legalMoves board diceRoll currSide
    forwardPruningMoves = forwardPruning board currSide allLegalMoves
pruningDepth
    minValuePar' :: (Ord t, Fractional t, NFData t) => [[Move]] -> t -> t
-> t -> t
    minValuePar' [] _ _ bestScore = bestScore
    minValuePar' (mv:mvs) al bt bestScore = case (performMoves board
currSide mv) of
      (Left _) -> minValuePar' mvs al bt bestScore
      (Right newBoard) -> if newBestScore <= al
                          then newBestScore
                          else minValuePar' mvs al newBt newBestScore where
        expectiRes = expectinodePar newBoard side (opposite currSide) al bt
(depth-1) pruningDepth
        newBestScore = min bestScore expectiRes
        newBt = min bt newBestScore

maxValuePar :: (Fractional t, Ord t, NFData t) => Board -> Side -> Side ->
Dice -> t -> t -> Int -> Int -> t
maxValuePar board side currSide diceRoll alpha beta depth pruningDepth
  | length allLegalMoves > 0 = maxValuePar' forwardPruningMoves alpha beta
(-1/0)
  | otherwise = expectinodePar board side (opposite currSide) alpha beta
(depth-1) pruningDepth where
    allLegalMoves = legalMoves board diceRoll currSide
    forwardPruningMoves = forwardPruning board currSide allLegalMoves
pruningDepth
    maxValuePar' :: (Ord t, Fractional t, NFData t) => [[Move]] -> t -> t
-> t -> t
    maxValuePar' [] _ _ bestScore = bestScore
    maxValuePar' (mv:mvs) al bt bestScore = case (performMoves board
currSide mv) of
      (Left _) -> maxValuePar' mvs al bt bestScore
      (Right newBoard) -> if newBestScore >= bt
                          then newBestScore
                          else maxValuePar' mvs newAl bt newBestScore where
```

```
        expectiRes = expectinodePar newBoard side (opposite currSide) al bt
(depth-1) pruningDepth
        newBestScore = max bestScore expectiRes
        newAl = max al newBestScore

main :: IO()
main = do
  args <- getArgs
  let side = read $ head args :: Side
  let restArgs = map (\x -> (read x :: Int)) (tail args)
  let depth = (restArgs !! 0)
  let pruningDepth = (restArgs !! 1)
  let seed = (restArgs !! 2)
  let ans = gamePlay side depth pruningDepth seed bestMovePar
  print $ ans
```

BackParNP.hs

```
{-
Parallel No Alpha-Beta pruning implementation of functions, uses
backgammon.hs module
Check Project report for explaination of each function.
-}

import Backgammon
import Control.Parallel.Strategies(using, parList, rdeepseq)
import Control.DeepSeq(NFData)
import Data.List(sortBy)
import System.Environment(getArgs)


bestMovePar :: (Eq a, Num a) => Board -> (Die, Die) -> Side -> a -> Int ->
[Move]
bestMovePar board diceRoll side depth pruningDepth = bestMovePar'
forwardPruningMoves where
  allLegalMoves = legalMoves board diceRoll side
  forwardPruningMoves = forwardPruning board side allLegalMoves
pruningDepth
  bestMovePar' mvs = snd $ head $ sortBy (\x y -> compare (fst x) (fst y))
(bestMoveParAll mvs)
  bestMoveParAll mvs = map innerFunc mvs `using` parList rdeepseq
  innerFunc mv = case (performMoves board side mv) of
    (Left _) -> (((1/0)::Double),mv)
    (Right upBoard) -> ((-1*expectiRes), mv) where
      expectiRes = expectinodePar upBoard side (opposite side)
((-1/0)::Double) (1/0) depth pruningDepth

expectinodePar :: (Eq a, Num a) => Board -> Side -> Side -> p1 -> p2 -> a
-> Int -> Double
expectinodePar board side _ _ _ 0 _ = fromIntegral $ eval board side
expectinodePar board side currSide alpha beta depth pruningDepth
  | side==currSide = sum $ sumAllDice minValuePar
  | otherwise = sum $ sumAllDice maxValuePar where
    sumAllDice func = map (\diceRoll -> (multiplier diceRoll) *
                                  func board side currSide diceRoll alpha
beta depth pruningDepth) allDiceRolls
                                  `using` parList rdeepseq
    multiplier (d1,d2) = if (d1==d2) then (1/36) else (1/18)
```

```haskell
minValuePar :: (Eq a, Num a) => Board -> Side -> Side -> (Die, Die) -> p1
-> p2 -> a -> Int -> Double
minValuePar board side currSide diceRoll alpha beta depth pruningDepth
  | length allLegalMoves > 0 = minValuePar' forwardPruningMoves
  | otherwise = expectinodePar board side (opposite currSide) alpha beta
(depth-1) pruningDepth where
    allLegalMoves = legalMoves board diceRoll currSide
    forwardPruningMoves = forwardPruning board currSide allLegalMoves
pruningDepth
    minValuePar' mvs = fst $ head $ sortBy (\x y -> compare (fst x) (fst
y)) (minValueParAll mvs)
    minValueParAll mvs = map innerFunc mvs `using` parList rdeepseq
    innerFunc mv = case (performMoves board currSide mv) of
      (Left _) -> (((1/0)::Double),mv)
      (Right newBoard) -> ((expectiRes), mv) where
        expectiRes = expectinodePar newBoard side (opposite currSide) alpha
beta (depth-1) pruningDepth

maxValuePar :: (Eq a, Num a) => Board -> Side -> Side -> (Die, Die) -> p1
-> p2 -> a -> Int -> Double
maxValuePar board side currSide diceRoll alpha beta depth pruningDepth
  | length allLegalMoves > 0 = maxValuePar' forwardPruningMoves
  | otherwise = expectinodePar board side (opposite currSide) alpha beta
(depth-1) pruningDepth where
    allLegalMoves = legalMoves board diceRoll currSide
    forwardPruningMoves = forwardPruning board currSide allLegalMoves
pruningDepth
    maxValuePar' mvs = fst $ head $ sortBy (\x y -> compare (fst x) (fst
y)) (maxValueParAll mvs)
    maxValueParAll mvs = map innerFunc mvs `using` parList rdeepseq
    innerFunc mv = case (performMoves board currSide mv) of
      (Left _) -> (((1/0)::Double),mv)
      (Right newBoard) -> ((-1*expectiRes), mv) where
        expectiRes = expectinodePar newBoard side (opposite currSide) alpha
beta (depth-1) pruningDepth

main :: IO()
main = do
  args <- getArgs
  let side = read $ head args :: Side
  let restArgs = map (\x -> (read x :: Int)) (tail args)
  let depth = (restArgs !! 0)
  let pruningDepth = (restArgs !! 1)
```

```
let seed = (restArgs !! 2)
let ans = gamePlay side depth pruningDepth seed bestMovePar
print $ ans
```

Acknowledgements

Code and Ideas borrowed from:

- Backgammon python implementation: https://github.com/chanddu/Backgammon-python-numpy-
- Backgammon haskell implementation: https://github.com/mmakowski/backgammon-model
  (This isn't fully functional)