Parallel Functional 2048

Colin Brown and Jonathan Rich

Fall 2019

1 Setup

To build, navigate to the project root directory. Then, run:

```
stack build
```

This will build an executable hidden in the .stack-work folder. To run the program, run:

```
stack exec <player type> <search depth>
```

The options for player type are:
int: Interactive player where you can play the game (just like the original game)
simple: A basic adversarial search agent without any parallelism or alpha-beta pruning
par: An agent that runs basic adversarial search in parallel across subtrees
ab: An agent that uses a sequential alpha-beta pruning algorithm
mixed: An agent that merges some parallelism and alpha-beta pruning using a "Young Brothers Can Wait" approach

2 Explanation of the Algorithms Implemented

All of the different algorithms are variants on a minimax algorithm to play 2048, modelling the random choices of where the next tile is placed as the intelligent decisions of an adversary.

simple: Implements a minimax algorithm. No parallelization or pruning is used.
ab: Implements an alpha-beta pruning algorithm. No parallelization is used.
par: Implements a minimax algorithm in parallel. This is a parallel version of the simple player.
mixed: Implements a minimax algorithm in parallel using a "Young Brothers Can Wait Strategy", which means exploring the first option at each level of the search tree in full, then using the alpha and beta values obtained from the first "Oldest Brother" branch to prune while running the remaining branches in parallel.

The mixed strategy is used due to the tradeoff between alpha-beta pruning, which speeds up the overall computation time required for the search but requires information to be propagated from one subtree to the next, vs parallelization, which splits the overall computation time across multiple processors but requires each subtree to be independent. The YBCW strategy gets some of the benefits of both, resulting in an overall faster algorithm.

3 Code Listing

The code is broken down into three files.

Main.hs: Entrypoint of the program. Parses command-line arguments (main), and contains the main game loop (playGame).
Base2048.hs: Implements functions and defines data types to simulate 2048 gameplay.
GameAgents.hs: Implements various different agents to play 2048. Some agents take advantage of parallel computation, and some do not.

The most important sections of each file are broken down below.

3.1 Main.hs

main: Parses command line arguments

```
main :: IO ()
main = do args <- getArgs
          case args of
            [maxdepth, playerType] -> playGame $ player (read maxdepth)
              where player = case playerType of
                        "simple" -> simplePlayer
                        "ab" -> alphaBetaPlayer
                        "par" -> fullParallelPlayer
                        "mixed" -> ybcwPlayer
                        p -> error $ p ++ " is not a valid player type"
            ["int"] -> playGame interactivePlayer
            _ -> do pn <- getProgName
                    die $ "Usage: " ++ pn ++ " <max depth> <player type>"
```

playGame: Given a player agent function, simulates a game of 2048 and prints the gameboard after each move.

```
playGame :: (GameState -> IO GameState) -> IO ()
playGame player = newGame >>= playFrom
  where playFrom g = do
            print g
            case g of
                (PlayerTurn _) -> player g >>= playFrom
                (ComputerTurn _) -> playComputer g >>= playFrom
                (GameWon _) -> putStrLn "Congratulations!" >> exitSuccess
                (GameLost _) -> putStrLn "Try again soon!" >> exitFailure
```

3.2 Base2048.hs

Constant declarations

```
size = 4
goal = 2048

computerChoices = [2, 4]
computerProbs = [0.9, 0.1]
```

instance Show GameState: determines how to print a GameState

```
instance Show GameState where
    show (PlayerTurn b) = "Player to Move.\n" ++ printBoard b
    show (ComputerTurn b) = "Computer to Move.\n" ++ printBoard b
    show (GameWon b) = "Player Won the Game!\n" ++ printBoard b
    show (GameLost b) = "Player Lost the Game!\n" ++ printBoard b

printBoard :: Board -> String
printBoard b = "\n" ++ (intercalate "\n" $ map showBoardRow b) ++ "\n"
    where showBoardRow r = intercalate "\t" $ map show r
```

newGame: represents a new game where the computer has played a single time

```
newGame :: IO GameState
newGame = playComputer $ ComputerTurn [replicate size 0 | _ <- replicate size 0]
```

scoreGame: a heuristic function for GameState

```
{-
    Heuristic for how good a state is for the player
    Rewards empty squares, higher value tiles, and evenly increasing rows/cols
    Penalizes large gaps in value between neighboring tiles
    Adds weighted rewards for values along a snaking path for easy combination
-}
scoreGame :: GameState -> Double
scoreGame (GameLost _) = -1 * (read "Infinity")::Double
scoreGame (GameWon _) = (read "Infinity")::Double
scoreGame (PlayerTurn board) = scoreBoard board
scoreGame (ComputerTurn board) = scoreBoard board

scoreBoard :: Board -> Double
scoreBoard b = 2^numEmpty + 0.5*sumPow + monoScore + snakeBias - 0.65*smoothPen
  where numEmpty = sum $ map (length . filter (0 ==)) b
        sumPow = sum $ map (sum . map (flip (**) (1.7::Double) . fromIntegral)) b
        smoothPen = fromIntegral $
            sumOverDirs (sum . map (sum . (map abs) . offsetDiffs)) b
        monoScore = fromIntegral $
            sumOverDirs (maximum . (map sum) . transpose . (map monoScores)) b
        monoScores x = map (abs . sum) $ tupToL $ partition (0<) $ offsetDiffs x
        offsetDiffs l = zipWith (-) l $ tail l
        sumOverDirs f b' = sum $ map f [b', transpose b']
        tupToL (x,y) = [x,y]
        snakeBias = fromIntegral $ weightSum $ concat $ flipAltRows
        weightSum l = sum $ zipWith (*) (map (flip (^) 3) [0..]) l
        flipAltRows = zipWith ($) (cycle [id, reverse]) b
```

nextStates: a wrapper to abstract turns and moves away from agents

```
nextStates :: GameState -> [GameState]
nextStates (PlayerTurn b) = map snd $ getPlayerMoves b
nextStates (ComputerTurn b) = getComputerMoves b
nextStates _ = []
```

getPlayerMoves: given a GameState, returns every possible legal tuple of move and resulting game state from playing that move.

```
getPlayerMoves :: Board -> [(Move, GameState)]
getPlayerMoves board = do
    move <- moves
    let successor = doPlayerMove board move
        state = getState successor
    guard $ successor /= board
    return (move, state)
  where moves = [LeftMove, RightMove, UpMove, DownMove]
        getState b | winState b = GameWon b
                   | otherwise  = ComputerTurn b
        winState b = any (>= goal) $ concat b
```

doPlayerMove: given a board and a player move, output the new board

```
-- fourfold symmetry of board, so implemented actual transformation only once
doPlayerMove :: Board -> Move -> Board
doPlayerMove b LeftMove = board'
  where board' = map (take size . (++ repeat 0) . merge . filter (/= 0)) b
        merge (x1:x2:xs)
            | x1 == x2 = (x1 * 2) : merge xs
            | otherwise = x1 : merge (x2 : xs)
        merge l = l
doPlayerMove b RightMove = map reverse $ doPlayerMove (map reverse b) LeftMove
doPlayerMove b UpMove = transpose $ doPlayerMove (transpose b) LeftMove
doPlayerMove b DownMove = transpose $ doPlayerMove (transpose b) RightMove
```

parsePlayerMove: used to convert string input to a move for interactive player

```
parsePlayerMove :: String -> Maybe Move
parsePlayerMove "L" = Just LeftMove
parsePlayerMove "R" = Just RightMove
parsePlayerMove "U" = Just UpMove
parsePlayerMove "D" = Just DownMove
parsePlayerMove _ = Nothing
```

getComputerMoves: given a board, returns a list of all possible GameStates after the computer plays

```
getComputerMoves :: Board -> [GameState]
getComputerMoves board = map getComputerState newBoards
  where newBoards = map snd $ allComputerMoveProbs board
```

playComputer: simulates a computer move on a GameState

```
playComputer :: GameState -> IO GameState
playComputer (ComputerTurn b) = doComputerMove b >>=
    \b' -> return $ getComputerState b'
playComputer _ = error "It's not the Computer's turn."
```

getComputerState: gets all possible successor boards when the computer plays

```
getComputerState :: Board -> GameState
getComputerState b | loseState b = GameLost b
                   | otherwise  = PlayerTurn b
  where loseState b' = gridMin b' > 0
            && minimum (map diffMin [b', transpose b']) > 0
        diffMin b' = minimum $ map (minimum . (map abs) . offsetDiffs) b'
        offsetDiffs l = zipWith (-) l $ tail l
        gridMin g = minimum (map minimum g)
```

doComputerMove: simulates a computer move on a board

```
doComputerMove :: Board -> IO Board
doComputerMove board = getStdRandom random >>=
    \r -> return $ snd $ head $ filter ((> r) . fst) $ cumulativeProbs
  where
    cumulativeProbs = scanl1 addCumulative $ allComputerMoveProbs board
    addCumulative = \(p, b) (p', b') -> (p + p', b')
```

allComputerMoveProbs: given a board, returns a list of tuples of probability and successor board, for every possible successor board after the computer plays. This function works by flattening the 2D board array, generating a list of all possible "deltas" (lists of zeros except for the computer's move), and then combining the two.

```
allComputerMoveProbs :: Board -> [(Double, Board)]
allComputerMoveProbs board = zip probs boards'
  where flatboard = concat board
        flatboards = replicate (size^2 * length(computerChoices)) flatboard
        flatboards' = zipWith combineDelta flatboards deltas
        uniqueflatboards' = filter (/= flatboard) flatboards'
        boards' = map (divvy size size) uniqueflatboards'

        probs = cycle $
                map (* (lenratio computerChoices uniqueflatboards'))
                computerProbs

        deltas = interleave $ map ndeltas computerChoices
        ndeltas n = (n : repeat 0) : (map (0:) $ ndeltas n)

        lenratio a b = (fromIntegral $ length a) / (fromIntegral $ length b)

        interleave = concat . transpose
        combineDelta flatboard delta = zipWith combine flatboard delta
        combine 0 new = new
        combine orig _ = orig
```

3.3 GameAgents.hs

The players frequently reuse code from other players; for example, the YCBM player (Mixed strategy) uses sequential alpha-beta search below the depth limit rather than parallelizing further.

Constants, Data Types, and Convenience Functions

```
inf = read "Infinity" :: Double
depthLimit = 2 :: Int

-- types to clean up function signatures
type MMResult = (Double, GameState)
type Minimax = Int -> GameState -> MMResult
type ABMinimax = Double -> Double -> Int -> GameState -> MMResult
type ABMap = Double -> Double -> Int -> [GameState] -> [MMResult]

fstMax = maximumBy (comparing fst) :: [MMResult] -> MMResult
fstMin = minimumBy (comparing fst) :: [MMResult] -> MMResult
```

simplePlayer: implements the "simple" Player

```
simplePlayer :: Int -> GameState -> IO GameState
simplePlayer maxdepth game = return $ snd $ seqMM maxdepth game
```

```
seqMM :: Minimax
seqMM 0 g                    = (scoreGame g, g)
seqMM d g@(PlayerTurn _)     = fstMax $ seqMMVals seqMM d $ nextStates g
seqMM d g@(ComputerTurn _)   = fstMin $ seqMMVals seqMM d $ nextStates g
seqMM _ g                    = (scoreGame g, g)
```

```
-- gets minimax values of next layer lazily
seqMMVals :: Minimax -> Int -> [GameState] -> [MMResult]
seqMMVals mm d gs = [(fst $ mm (d-1) x, x) | x <- gs]
```

fullParallelPlayer: implements the "par" Player

```
fullParallelPlayer :: Int -> GameState -> IO GameState
fullParallelPlayer maxdepth game = return $ snd $ parMM maxdepth game
```

```
parMM :: Minimax
parMM 0 g                    = (scoreGame g, g)
parMM d g@(PlayerTurn _)     = fstMax $ seqMMVals parMM d $ nextStates g
parMM d g@(ComputerTurn _)   = fstMin $ parMMVals parMM seqMM d $ nextStates g
parMM _ g                    = (scoreGame g, g)
```

```
-- gets minimax values of next layer in parallel
-- uses parmm minimax function in parallel if above a certain depth
-- otherwise finishes tree using seqmm minimax function
parMMVals :: Minimax -> Minimax -> Int -> [GameState] -> [MMResult]
parMMVals parmm seqmm d gs | d > depthLimit  = zipWith (,) scores gs
                           | otherwise       = seqMMVals seqmm d gs
  where scores = map fst $ parMap rpar (parmm (d-1)) gs
```

alphaBetaPlayer: implements the "ab" player

```
alphaBetaPlayer :: Int -> GameState -> IO GameState
alphaBetaPlayer maxdepth game = return $ snd $ abMM (-1*inf) inf maxdepth game
```

```
abMM :: ABMinimax
abMM _ _ 0 g                   = (scoreGame g, g)
abMM a b d g@(PlayerTurn _)    = fstMax $ maxABEval a b d $ nextStates g
abMM a b d g@(ComputerTurn _) = fstMin $ minABEval a b d $ nextStates g
abMM _ _ _ g                   = (scoreGame g, g)
```

```
-- functions implementing the AB updates and pruning
maxABFold :: ABMinimax -> ABMap
    -> Double -> Double -> Int -> [GameState] -> [MMResult]
maxABFold mm _ a b d [g]                        = [(s, g)]
  where s = fst $ mm a b (d-1) g
maxABFold mm eval a b d (g:gs) | newA >= b = [(s, g)]
                               | otherwise = (s, g) : eval newA b d gs
  where s = fst $ mm a b (d-1) g
        newA = max a s

minABFold :: ABMinimax -> ABMap
    -> Double -> Double -> Int -> [GameState] -> [MMResult]
minABFold mm _ a b d [g]                        = [(s, g)]
  where s = fst $ mm a b (d-1) g
minABFold mm eval a b d (g:gs) | a >= newB = [(s, g)]
                               | otherwise = (s, g) : eval a newB d gs
  where s = fst $ mm a b (d-1) g
        newB = min b s

-- Map a list of gameStates using sequential propagation of AB values
maxABEval :: ABMap
maxABEval a b = maxABFold abMM maxABEval a b
minABEval :: ABMap
minABEval a b = minABFold abMM minABEval a b
```

ycbwPlayer: implements the "mixed" player

```
ybcwPlayer :: Int -> GameState -> IO GameState
ybcwPlayer maxdepth game = return $ snd $ ybcwMM (-1*inf) inf maxdepth game
```

```
ybcwMM :: ABMinimax
ybcwMM _ _ 0 g                   = (scoreGame g, g)
ybcwMM a b d g@(PlayerTurn _)    = fstMax $ maxYBCWEval a b d $ nextStates g
ybcwMM a b d g@(ComputerTurn _) = fstMin $ minYBCWEval a b d $ nextStates g
ybcwMM _ _ _ g                   = (scoreGame g, g)
```

```
-- Map a list of gameStates using by mapping in parallel until the depthLimit
parABEval :: ABMap
parABEval a b = parMMVals (ybcwMM a b) (abMM a b)
```

```
-- Map a list by getting ab values from the first and then mapping in parallel
maxYBCWEval :: ABMap
maxYBCWEval a b = maxABFold ybcwMM parABEval a b
minYBCWEval :: ABMap
minYBCWEval a b = minABFold ybcwMM parABEval a b
```

interactivePlayer: implements the "int" player (human controlled)

```
interactivePlayer :: GameState -> IO GameState
interactivePlayer g@(PlayerTurn b) = do
    putStrLn "Choose Move (L, R, U, D)"
    m <- parsePlayerMove <$> getLine
    case m of
        Just m' -> do
            case lookup m' $ getPlayerMoves b of
```

```
                  Just newState -> return newState
                  Nothing -> getAnotherMove
         Nothing -> getAnotherMove
  where getAnotherMove = do putStrLn "Move invalid! Choose another move."
                            interactivePlayer g
interactivePlayer _ = error "It's not the Player's turn"
```

4 Results

These test suites are convenience scripts that invokes the Haskell modules several times with various options. Python 3 is required to run the test script. Certain options like suppressing program output don't work on Windows (if possible, run on Linux or MacOS).

4.1 gametest.py

This test script runs each of the 4 player types calculates the average time taken to win a game for each of the 4 player types for a specified search depth. It can be run from the project root directory using:

```
python3 test/gametest.py <number of iterations> <maximum depth> <show games>
```

Recommended values for test script parameters:
number of iterations: 1 - 3. The script will run until each algorithm has successfully solved the puzzle this many times before completing.
maximum depth: 4 - 5. The player agents will attempt to search the game tree up to this depth at a maximum.
show games: true — false. If true, the script will print the output of each game as it is played.

Note that the test suite may take a long time to run (10+ minutes) and need to retry several attempts in the course of its run. This is normal.
This test script was run on MacOS High Sierra with a 2.2 GHz Intel Core i7 processor and 16GB of RAM. The machine has 4 physical cores and 8 logical cores. The machine had minimal background programs and processes running for the duration of the test.

The test averaged 20 successful iterations of the program for each algorithm, with a maximum depth of 5, using the command:

```
python3 test/test.py 20 5 false
```

Algorithm Results:

```
simple: Average successful runtime 62.85371502637863, Success Rate 36.36363636363637
ab: Average successful runtime 12.893631625175477, Success Rate 21.73913043478261
par +RTS -N -s: Average successful runtime 15.325394880771636, Success Rate 39.21568627450981
mixed +RTS -N -s: Average successful runtime 6.3631915807723995, Success Rate 40.0
```

Using simple as a benchmark, we find:

ab: 4.87x faster for successful runs
par: 4.10x faster for successful runs
mixed: 9.88x faster for successful runs

We note that the success rate (the percentage of games that the agent wins) is comparable for all of the algorithms!

4.2 searchtest.py

This script is used to test the average time taken to complete a search on a predetermined set of gamestates, which provides a more consistent estimation of speedups of the search agents (as well as ignoring the extra time taken in playing the game for the computer to take its turns).

This python script uses the compiled source file produced from Spec.hs (located in the test folder), which takes in a list of boards in a text file (boards.txt) and solved each one in turn using the provided player type and search depth. This file can be compiled from the test directory using the command:

```
stack build && stack ghc -- -make -O2 - threaded -rtsopts -eventlog -O spec
```

and the script itself can be run with the command:

```
python3 searchtest.py <number of iterations> <maximum depth> <show games>
```

The parallel algorithms can be run with a number of variations to control the number of created sparks, corresponding to the depth limit below which search is run sequentially as well as only parallelizing on min or max rows rather than both. This can be done by changing the depthlimit argument or modifying **minYBCWEval** to **minABFold ybcwMM minABEval** (with the equivalent for max).
The results of running each variation for the best performing algorithm, Mixed, with 10 iterations and a search depth of 5 are as follows (all using the max number of cores):

Average Times for Mixed:

depth limit	both	max	min
0	6.17	8.87	9.876
1	6.03	8.89	9.81
2	5.52	8.86	9.79
3	5.07	8.80	9.97
4	12.61	12.83	10.19

The fastest parallel variation for mixed was using full parallelization for all levels above depth 3, which gives a speedup of 7.76x relative to the baseline (recalculated by running searchtest using simple).