Parallelizing 2048

Matthew Broughton - mb4207, Kent Hall - kjh2166

December 2021

1 Background - What is 2048?

2048 is a game where players move tiles valued at powers of 2 around a 4x4 grid while trying create a tile of value 2048. Every turn, a 2 or 4 valued tile appears on the board. The player can then shift the tiles in one of 4 directions. The tiles move until they hit the edge of the board or another tile. If two tiles collide and both are the same value, they merge and the resulting tile has double the value. In this fashion the player can manipulate the tiles to create as many high value tiles as possible. While the game is considered won when a tile of value 2048 is created, the player can continue playing the game until no possible moves are left. Play here https://play2048.co/ (it's fun).

2 Base Algorithm - Expectiminimax

The optimal algorithm for an AI to run off of to try and win a game of 2048 is expectiminimax. Expectiminimax is a variation of minimax, which is a adversarial algorithm for choosing the next move in a game. This incorrectly assumes that the computer is intentionally placing new tiles in antagonistic ways against the player, but the algorithm still has the same result of picking the optimal move (most of the time). In minimax, the algorithm creates a decision tree where each level represents the player's or the computer's turn (alternating), and each node represents an action. Each node contains a value corresponding to how good the board is for the player, computed via some heuristic. The algorithm then finds the optimal move for the current turn by searching the tree (via DFS) for the move that not only maximizes the board's value now, but also maximizes it over the course of the next X turns while also minimizing the possible damage by the computer over the same time frame. Expectiminimax adds in an additional level to the tree that simulates random element, for our purposes it'll simulate whether a 2 or 4 tile is placed.

The heuristics we use to decide whether or not a board is good are:

- Smoothness - The more tiles that have neighboring tiles of the same value, the better the board is. Tiles can only merge if they have the same value, so this heuristic promotes mergeability.

- Monotonicity - The more rows/columns that are monotonically decreasing from left to right/up to down, the better the board is. This keeps the board ordered and larger tiles off to the left/upper sides, allowing smaller tiles to more easily merge in the open bottom right.
- Weight - Arbitrary weights are multiplied with the values of each tile, so the more higher valued tiles there are, the better the board is. Promotes having bigger tiles, and the weights are arranged in such a way (with the largest weight in the upper left corner) to promote monotonicity as well.
- Number of Empty Spaces - The fewer tiles there are, the better the board is. Promotes a clutter-free board.
- 2048 - If a possible board has 2048 on it, it gets an extremely high score. Promotes winning the game.

This algorithm is essentially just a lengthy tree traversal, meaning that the time taken increases exponentially with each level added to the tree. Since deeper trees means a higher win rate, speeding up this traversal is our top priority.

3 Sequential Optimization - AB Pruning

While the focus of this paper is speed-ups via parallelization, there is a well-known, sequential, optimization for this algorithm. This is Alpha-Beta pruning, which removes branches of the tree that cannot possibly be optimal from consideration as the algorithm runs. We provide a well put explanation from Wikipedia: "The algorithm maintains two values, alpha and beta, which respectively represent the minimum score that the maximizing player is assured of and the maximum score that the minimizing player is assured of. Initially, alpha is negative infinity and beta is positive infinity, i.e. both players start with their worst possible score. 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."

Something to note is that this algorithm can not be parallelized without significant editing. Since the Alpha/Beta pair is passed between sibling nodes from the parent node, you are not able to traverse down separate child nodes in parallel without resetting the Alpha/Beta pair, which limits the amount of pruning you can perform and thus the amount of speed-ups you will gain from this algorithm.

While we will focus on mostly on speed-ups over the base Expectiminimax algorithm, we will also compare to AB Pruning. We will also incorporate limited AB Pruning into our parallelization algorithms, as you will see in the next section.

4 Parallel Optimization - parMap and Brothers

The obvious place to start parallelizing is with the base algorithm. Since nothing passes between child nodes from a parent node, we can, for each parent, perform all traversals from their children down in parallel. For example, in implementation at a maximize node, we can simply call ‘parMap rpar (minimize) [list of child nodes]‘. Naively, we could do this for every maximize, minimize, and chance node in the tree, but too much parallelization may create too much overhead. Thus we may need to limit how deep our parallelization goes in the tree. This will be our base parallelization algorithm, which we will call Par+Base.

The logical follow up to this algorithm is adding AB Pruning to it. While we cannot simply add pruning to entire tree as discussed before, what we can do is go parallel to a certain depth, then once the parallelization stops, we can perform pruning on the available subtrees. While there won't be as significant pruning as there could be if we recieved Alpha/Beta values from the tree's root, we hope that our limited pruning will cause enough of a speed-up to blaze past the standard sequential AB Pruning. We will call this version Par+AB.

Admittedly, Par+AB is a rather simple way of combining AB Pruning and parallelization. What if there was a better way that leverages the power of AB Pruning a little more effectively? Here, we present the Younger Brothers Wait Concept, which we will call Brothers for short. First found here (https://www.chessprogramming.org/Young_Brothers_Wait_Concept), Brothers changes how all the nodes are evaluated slightly. First, one child node (the 'eldest') is chosen to be fully evaluated with no parallelization whatsoever. Once this child node is evaluated, the Alpha/Beta bounds for the remaining children are sharpened significantly. We can then evaluate the remaining ('younger') children in parallel with the tighter bounds. Like before, we can go parallel to a set depth here to ensure that we do not get overwhelmed by overhead.

5 Parallel Evaluation Metrics - Spark Conversion

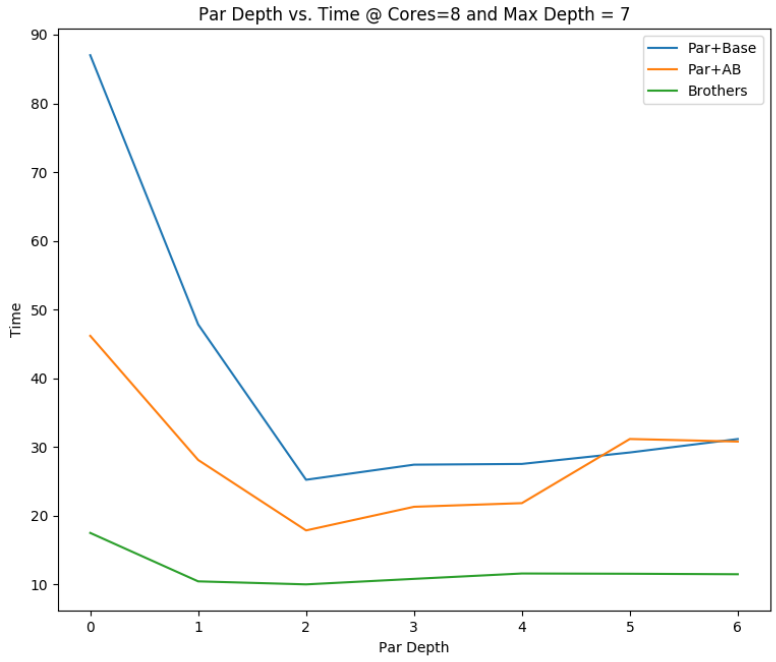
The best way to evaluate all of these algorithms is clearly by time, as this is a race to evaluate possible moves the fastest. However, for our parallel algorithms, we have a few more things we can measure on. Chief among these is the percentage of converted sparks. A spark is converted when it completes its task; an unconverted spark is (justly) killed prior to task completion, meaning that any work it performed was wasted and better spent on a spark that could be converted. While time is the ultimate judge of how worthwhile our parallel algorithms are, the spark conversion rate can point to where our implementation could possibly be improved.

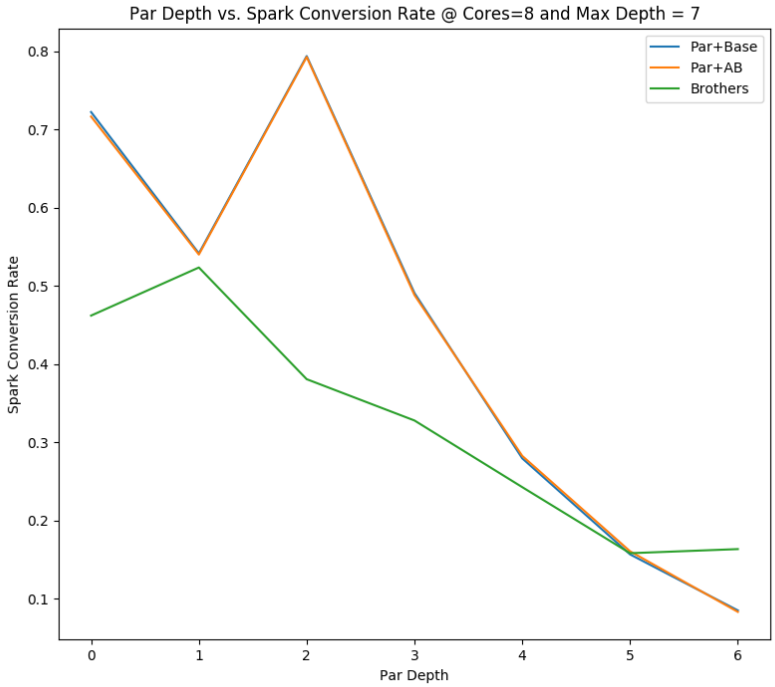
6 Results - Tuning Parallel Parameters

Here, we present preliminary results for the parallelized algorithms so we can ascertain the best parameters to use for each of them before stacking them up against the sequentials. The two parameters are Par Depth and Number of Cores. Par Depth is how many levels of the tree we have parallelized for a given run, and Number of Cores is of course how many cores we run the AI on.

6.1 Par Depth

We ran the AI with varying levels of par depth from 0 to the max depth over around 100 trials for each level and measured time and spark conversion rate. We set the number of cores to 8 and the max depth to 7 for all trials.

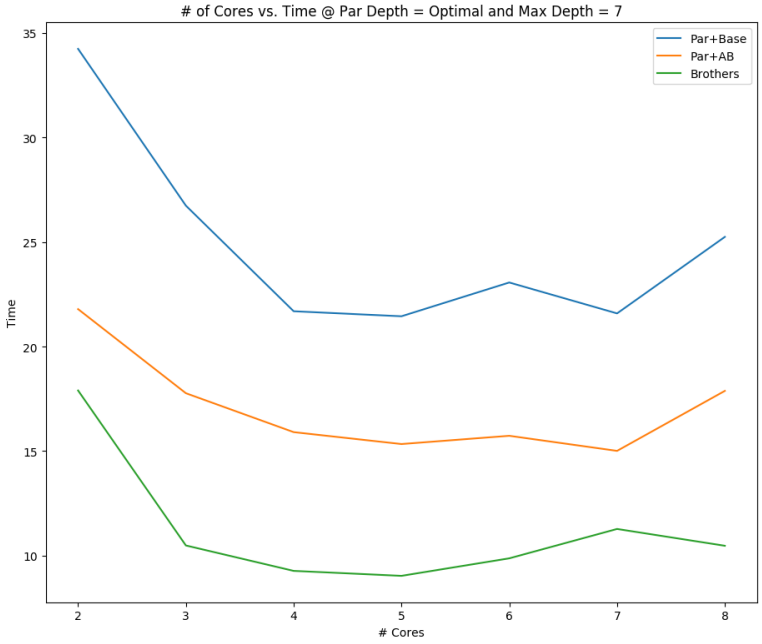


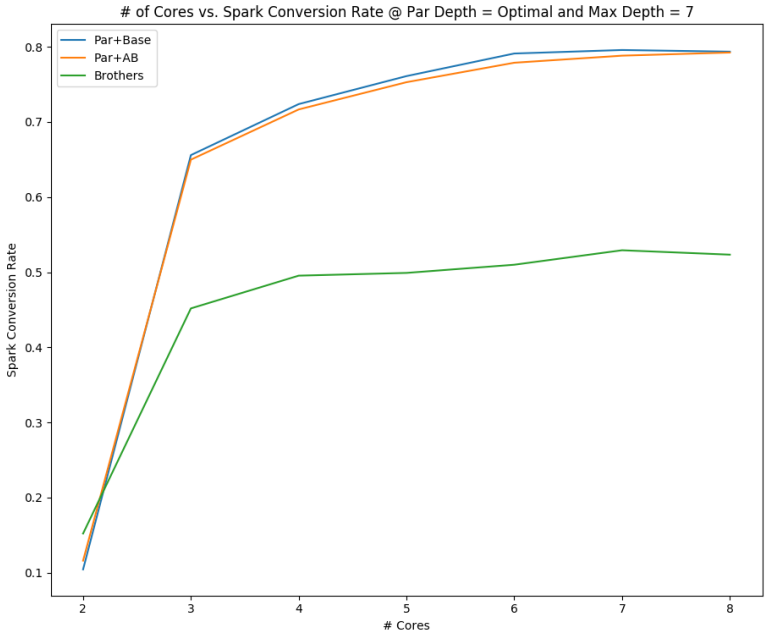


From both graphs, we can clearly see for Par+Base and Par+AB that 2 levels of par depth is clearly the fastest and most efficient. For Brothers, however, all of the times for each par depth (aside from 0) are fairly close together around 10 seconds, with par depth 2 being the lowest at 10.06sec. We only see a real breakaway on the spark conversion graph, where a depth of 1 has the highest rate at around 52%. While 52% is not terribly efficient as is, especially when compared to Par+Base and Par+AB, we will choose 1 as the optimal par depth for this algorithm.

6.2 Number of Cores

We ran the AI with 1 to 8 cores over around 100 trials for each core and measured time and spark conversion rate. We set the par depths to the values found in the previous section and the max depth to 7 for all trials.

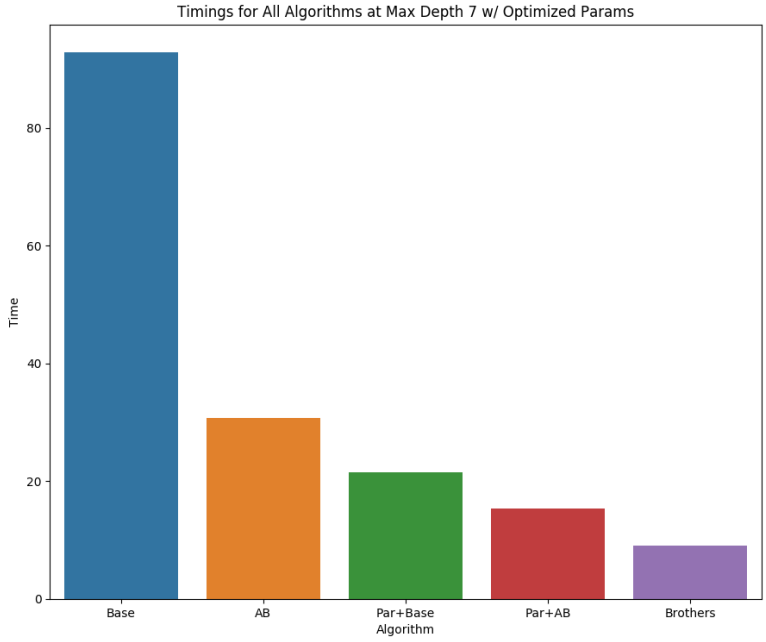

# of Cores vs. Time @ Par Depth = Optimal and Max Depth = 7
Par+Base
Par+AB
Brothers
Time
Cores
35
30
25
20
15
10
2
3
4
5
6
7
8



nCores = 5

7 Final Results

With ideal parameters for each algorithm, we find that basic Expectiminimax sees the greatest benefit from parallelization, with roughly a 4x speed improvement. AB Pruning improves this algorithm substantially in sequential execution, so the improvement from parallelization is more modest, but not insignificant (at about 2x). Incorporating the Younger Brothers Wait Concept gets us our fastest results, improving on parallelized AB Pruning even further. In any case, it's clear that additional threads of concurrent execution go a long way toward winning 2048.



Base	92.95407920792074 s
AB	30.68809677419355 s
Par+Base	21.447784313725496 s
Par+AB	15.334137254901966 s
Brothers	9.028725490196079 s

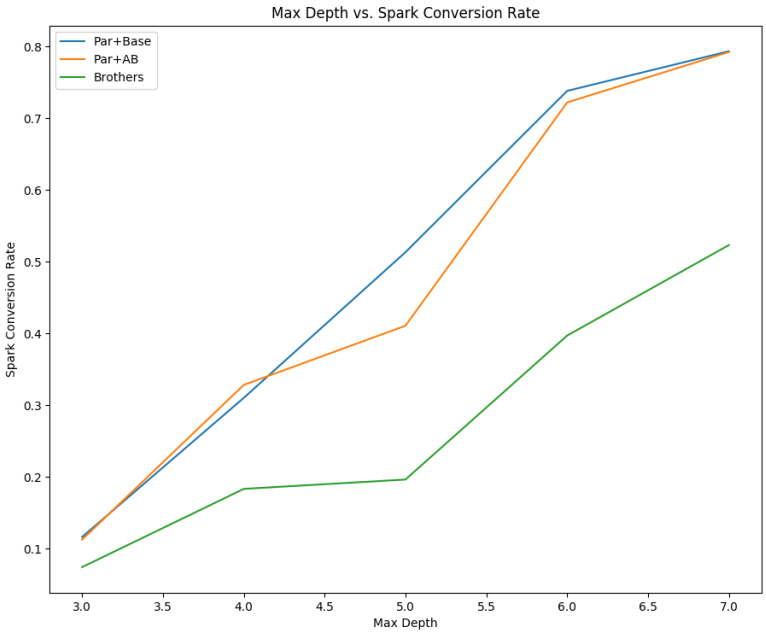
8 Further Work

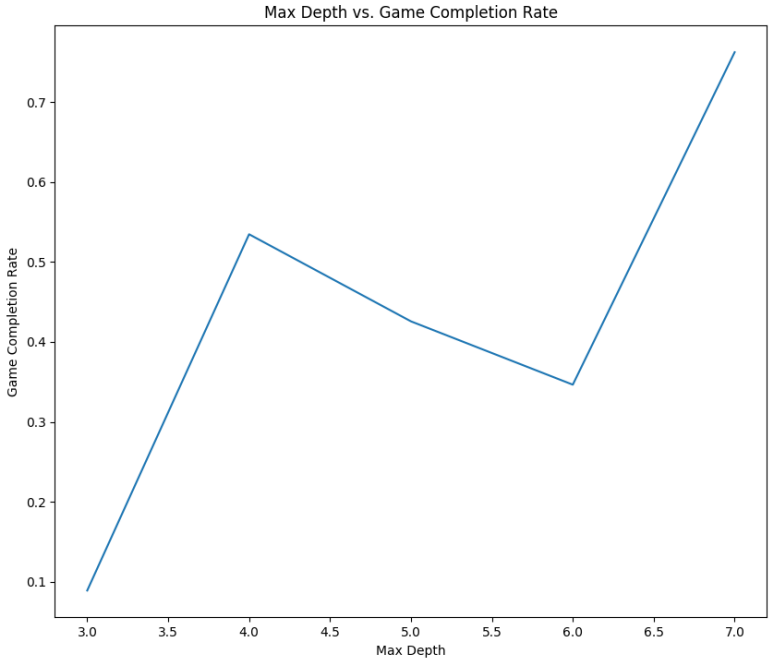
One lingering mystery is the question of why Brothers peaks at a 50% spark conversion rate, as opposed to Par+Base and Par+AB hitting 80% spark conversion. This is despite the fact that, at the end of the day, Brothers performs significantly better than both. We believe this could indicate there is additional headroom whereby further parallelization performance gains might be attainable, but this is uncertain; at a glance, it seems as though the nature of the algorithm might fundamentally be limiting how effective the parallelization is, but this is definitely a point of inquiry to look into in the future.

9 Max Depth

The focus of this report is on the performance implications of parallelization, so we didn't find it relevant to include details on our experimentation in finding

the optimal max depth; that said, the data is included below for completeness.





10 Code Listing

10.1 AlphaBetaAi.hs

```
module AlphaBetaAi where

import Board
import Heuristics

minInt :: Double
minInt = 0
maxInt :: Double
maxInt = 1073741824
defaultAB :: (Double, Double)
defaultAB = (minInt, maxInt)

abPlayerMove :: Int -> Board -> Board
abPlayerMove maxD b = snd $ abMaximize (0,maxD) defaultAB b

alphaBetaMax :: (Int,Int) -> Board -> ([Double], Board) -> ([Double], Board)
```

```
alphaBetaMax (d,maxD) board ([a, b, prevOut], prevBoard)
    | localMax > a  = ([localMax, b, localMax], localBoard)
    | otherwise     = ([a, b, localMax], localBoard)
    where score = if prevOut < b
                  then abChanceTime (d+1,maxD) (a,b) board
                  else minInt
          localMax = if score >= prevOut then score else prevOut
          localBoard = if score >= prevOut then board else prevBoard
alphaBetaMax _ _ _ = ([],[])

abMaximize :: (Int,Int) -> (Double, Double) -> Board -> (Double, Board)
abMaximize (d,maxD) (a,b) board
    | d >= maxD = (calcScore board, board)
    | otherwise =
        let f = foldr (alphaBetaMax (d,maxD)) ([a, b, minInt], []) in
        let ([_, _, lMax], localBoard) = f $ getAvailableMoves board in
        if lMax == minInt then (calcScore board, board) else (lMax, localBoard)

abChanceTime :: (Int,Int) -> (Double, Double) -> Board -> Double
abChanceTime (d,maxD) ab board
    | d >= maxD = calcScore board
    | otherwise = let f = (abMinimize (d+1,maxD) ab board) in
                  sum $ zipWith (*) [0.9, 0.1] $ map f [2,4]

alphaBetaMin :: (Int,Int) -> Board -> [Double] -> [Double]
alphaBetaMin (d,maxD) board [a, b, prevOut]
    | localMin < b  = [a, localMin, localMin]
    | otherwise     = [a, b, localMin]
    where score = if prevOut > a
                  then fst $ abMaximize (d+1,maxD) (a,b) board
                  else maxInt
          localMin = if score <= prevOut then score else prevOut
alphaBetaMin _ _ _ = []

abMinimize :: (Int,Int) -> (Double, Double) -> Board -> Tile -> Double
abMinimize (d,maxD) (a,b) board tile
    | d >= maxD = calcScore board
    | otherwise =
        let nxt = map (cpuMoveDet board tile) $ getAvailableTileIndices board in
        if length nxt == 0 then calcScore board else
        (foldr (alphaBetaMin (d,maxD)) [a, b, maxInt] nxt) !! 2
```

10.2 BaseAi.hs

```
module BaseAi where
```

```
import Board
import Heuristics
import AlphaBetaAi
import Control.Parallel.Strategies

basePlayerMove :: Bool -> (Int,Int) -> Board -> Board
basePlayerMove isAB (parD,maxD) bd = snd $ maximize isAB (0, parD, maxD) bd

maximize :: Bool -> (Int, Int, Int) -> Board -> (Double, Board)
maximize isAB (d, parD, maxD) board
    | d >= maxD = (calcScore board, board)
    | otherwise =
        let moves = getAvailableMoves board in
        let scoreB = map' (\bd -> (ct' bd, bd)) moves in
        if length moves == 0 then (calcScore board, board) else
        foldr (\x acc -> if ((fst x) > (fst acc)) then x else acc) ((-1),[]) scoreB
    where map' = if d > parD then map else parMap rpar
          ct'  = if (d >= parD) && isAB then abChanceTime (d+1,maxD) defaultAB else chanceTi

chanceTime :: Bool -> (Int, Int, Int) -> Board -> Double
chanceTime isAB (d, parD, maxD) board
    | d >= maxD = calcScore board
    | otherwise =
        let f = (min' board) in
        sum $ zipWith (*) [0.9, 0.1] $ map' f [2,4]
    where map' = if d > parD then map else parMap rpar
          min' = if (d >= parD) && isAB then abMinimize (d+1,maxD) defaultAB else minimize i

minimize :: Bool -> (Int, Int, Int) -> Board -> Tile -> Double
minimize isAB (d, parD, maxD) board tile
    | d >= maxD = calcScore board
    | otherwise =
        let nxt = map (cpuMoveDet board tile) $ getAvailableTileIndices board in
        let scoreB = map' (\bd -> fst $ max' bd) nxt in
        if length nxt == 0 then calcScore board else minimum scoreB
    where map' = if d > parD then map else parMap rpar
          max' = if (d >= parD) && isAB  then abMaximize (d+1,maxD) defaultAB else maximize
```

10.3 Board.hs

```
module Board where

import System.Random
import Data.List (transpose)

type Tile = Int
```

```
type Board = [Tile]
type TwoDBoard = [[Tile]]

getAvailableTileIndices :: Board -> [Int]
getAvailableTileIndices board =
    snd $ unzip $ filter (\x->fst x == 0) $ zip board $ take 16 $ iterate (+1) 0

-- places a 2 or 4 tile in a random availabe location
cpuMoveRng :: RandomGen rng => Board -> rng -> Board
cpuMoveRng board gen =
    [if t2 == idx then val else t1 | (t1,t2) <- zip board $ take 16 $ iterate (+1) 0]
    where
        idx = (getAvailableTileIndices board) !! ridx
        ridx = fst $ randomR (0,(length $ getAvailableTileIndices board)-1) gen
        val = if (fst $ randomR (0, 9 :: Int) gen)==0 then 4 else 2

cpuMoveDet :: Board -> Int -> Int -> Board
cpuMoveDet board val idx=
    [if t2 == idx then val else t1 | (t1,t2) <- zip board $ take 16 $ iterate (+1) 0]


-- rowMap implemenation stolen from splitEvery
rowMap :: Board -> TwoDBoard
rowMap [] = []
rowMap b = let (row,rst) = splitAt 4 b in row : rowMap rst

mergeTiles :: [Tile] -> [Tile]
mergeTiles [] = []
mergeTiles (hd:nx:tl) | hd == nx = mergeTiles ((2*hd):tl)
mergeTiles (hd:tl) | otherwise = hd : mergeTiles tl

move :: Board -> Int -> Board
move b dir
    | dir == 0 = columnUnmap $ map (mergeMaster) $ columnMap b
    | dir == 1 = columnUnmap $ map (reverse.mergeMaster.reverse) $ columnMap b
    | dir == 2 = rowUnmap $ map (mergeMaster) $ rowMap b
    | otherwise = rowUnmap $ map (reverse.mergeMaster.reverse) $ rowMap b
    where zFilt = filter (/=0)
          pad l = l ++ replicate (4 - (length (filter (/=0) l))) 0
          mergeMaster = pad.mergeTiles.zFilt
          rowUnmap = concat
          columnMap = transpose.rowMap
          columnUnmap = rowUnmap.transpose

-- up down left right
getAvailableMoves :: Board -> [Board]
```

```
getAvailableMoves b = filter (/=b) $ map (move b) [0,1,2,3]

printBoard :: Board -> IO ()
printBoard b = do
    let rb = rowMap b
    putStrLn "////////////////////"
    putStrLn $ show (rb!!0)
    putStrLn $ show (rb!!1)
    putStrLn $ show (rb!!2)
    putStrLn $ show (rb!!3)
    putStrLn "////////////////////"
```

10.4 Heuristics.hs

```
module Heuristics where

import Board
import Data.List (transpose, foldl')

-- these numbers came to me in a dream
calcScore :: Board -> Double
calcScore b = (1   * (smoothness b) +
              0.15 * (monotonicity b) +
              1.2  * (fromIntegral $ length $ getAvailableTileIndices b) +
              0.25 * (logBase 2 $ weight b) +
              9999 * (greatSuccess b))

smoothC :: [Tile] -> Int -> Int
smoothC (f:s:tl) acc | f == s    = smoothC (s:tl) (acc+1)
smoothC (_:s:tl) acc | otherwise = smoothC (s:tl) acc
smoothC _ acc = acc

smoothness :: Board -> Double
smoothness b = fromIntegral $
               (foldr smoothC 0 $ rowMap b) +
               (foldr smoothC 0 $ (transpose.rowMap) b)

gintonic :: (Int,Int) -> [Tile] -> (Int,Int)
gintonic (tot,bon) (f:s:tl) | f >= s = gintonic (tot+bon,bon+1) (s:tl)
gintonic acc (f:s:_) | f < s = acc
gintonic acc _ = acc

monotonicity :: Board -> Double
monotonicity b = fromIntegral $
                 (fst $ foldl' gintonic  (0,0) $ rowMap b) +
                 (fst $ foldl' gintonic  (0,0) $ (transpose.rowMap) b)
```

```
weight :: Board -> Double
weight b = fromIntegral $
           foldr (+) 0 $ zipWith (*) b $ map weightMap $ wHeatmap (0::Int)
           where wHeatmap n
                    | n == 4 = []
                    | otherwise = (map (n+) [0,1,2,3]) ++ (wHeatmap (n+1))
                 weightMap n
                    | n == 0 = 1000000
                    | n == 1 = 1000
                    | n == 2 = 100
                    | n == 3 = 10
                    | otherwise = 1

greatSuccess :: Board -> Double
greatSuccess b
    | maximum b == 2048 = 99999999
    | otherwise         = 0
```

10.5 KyoudAi.hs

```
module KyoudAi where

import Board
import AlphaBetaAi
import Heuristics
import Control.Parallel.Strategies

brotherMove :: (Int,Int) -> Board -> Board
brotherMove (parD,maxD) bd = snd $ maximize (0, parD, maxD) defaultAB bd

maximize :: (Int, Int, Int) -> (Double, Double) -> Board -> (Double, Board)
maximize (d, parD, maxD) (a,b) board
    | d >= maxD = (calcScore board, board)
    | otherwise = if length bds == 0 then (calcScore board, board) else
                  getScores
    where
        bds = getAvailableMoves board
        s = chanceTime (d+1,parD,maxD) (a,b) $ head bds
        getScores
            | s >= b = (s, head bds)
            | s > a = accum (s,b)
            | otherwise = accum (a,b)
        accum ab =
            let scoreB = (s, head bds) : (parMap rseq (\bd -> (ct' ab bd, bd)) $ tail bds) i
            foldr (\x acc -> if ((fst x) > (fst acc)) then x else acc) ((-1),[]) scoreB
```

```
        ct' = if (d > parD) then abChanceTime (d+1,maxD) else chanceTime (d+1,parD,maxD)

chanceTime :: (Int, Int, Int) -> (Double, Double) -> Board -> Double
chanceTime (d, parD, maxD) ab board
    | d >= maxD = calcScore board
    | otherwise = let f = (min' ab board) in
                  sum $ zipWith (*) [0.9, 0.1] $ parMap rseq f [2,4]
    where min' = if d > parD then abMinimize (d+1,maxD) else minimize (d+1,parD,maxD)

minimize :: (Int,Int,Int) -> (Double, Double) -> Board -> Tile -> Double
minimize (d, parD, maxD) (a,b) board tile
    | d >= maxD = calcScore board
    | otherwise = if length bds == 0 then calcScore board else
                  getScores
    where
        bds = map (cpuMoveDet board tile) $ getAvailableTileIndices board
        s = fst $ maximize (d+1,parD,maxD) (a,b) $ head bds
        getScores
            | s <= a = s
            | s < b = accum (a,s)
            | otherwise = accum (a,b)
        accum ab =
            let scoreB = s : (parMap rseq (\bd -> fst $ max' ab bd) $ tail bds) in
            minimum scoreB
        max' = if d > parD then abMaximize (d+1,maxD) else maximize (d+1,parD,maxD)
```