Parallelizing 2048 on Haskell

Unal Yigit Ozulku (uyo2000)

Goal of the project:

The goal of this project is to implement the minimax algorithm in Haskell to develop an agent that plays the game 2048, followed by coming up parallelization strategies to speed up the process

Algorithm:

This project implements the minimax algorithm with a custom heuristic function based on the number of empty tiles that are left in the game board. We then focus on parallelizing the minimax algorithm and analyzing metrics to understand the difference in performance from a sequential implementation to a parallel one.

Performance Gains:

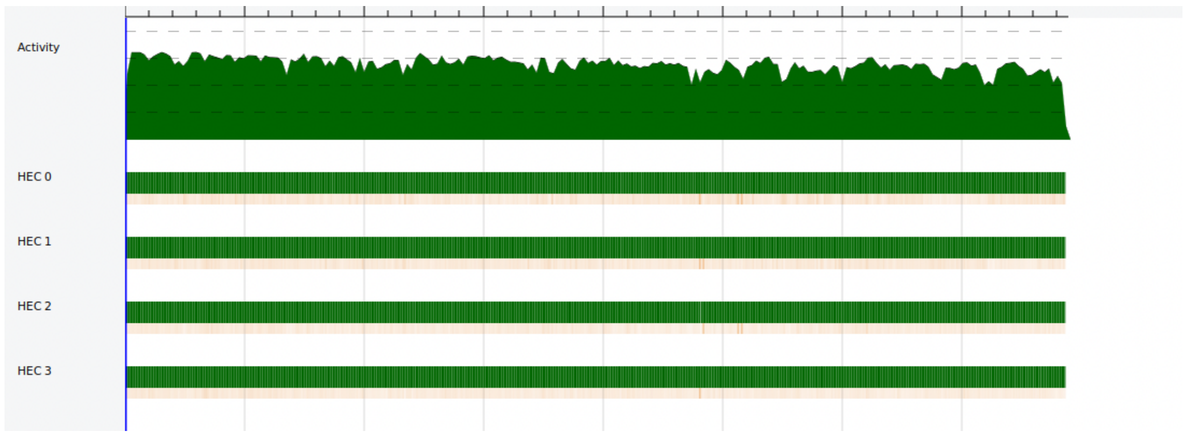
The code allows us to parallelize the minimax algorithm with different combinations of strategies: Strategy 1 (highlighted in the getMove function definition) allows a shallow parallelization, while Strategy 2 (highlighted in the score method) makes the parallelization more granular and allows for better load balancing.

```
getMove :: Int -> Board -> Direction
-- parList rseq -- Parallel
-- rseq -- Sequential
getMove depth board = parMaximumOn ([STRATEGY 1]) (score depth . flip slide board)
[minBound..maxBound]

score :: Int -> Board -> Int
score 0 board = length (emptyTiles board)
score depth board =
  case possibleNewTiles board of
    [] -> 0
    ts -> minimum (map (\b -> maximum (map (score (depth - 1) . flip slide b)
[minBound..maxBound] `using`[STRATEGY 2])) ts)

parMaximumOn :: (Ord b) => Strategy [(b,a)] -> (a -> b) -> [a] -> a
parMaximumOn strat f xs = snd (maximumBy (comparing fst) (map pairElementsWithScores xs
`using` strat))
  where
  pairElementsWithScores x = let w = f x in w `seq` (w, x)
```

Using our multithreaded parallel implementation (at depth = 3), we received the numbers below:

Cores	Time (s)	Moves	Time/move (s)
1	769.96	989	0.777
2	506.04	946	0.534
3	439.00	1002	0.438
4	395.22	1098	0.359

Due to the random component of 2048, some runs may take more moves to reach 2048 than others; therefore, the more telling metric is how much time did the solver take per move. As can be seen by the table, there is a significant time gain over the single-threaded version when we start parallelizing the game, and the time/move metric decreases slightly with each additional core. We also analyzed the load balancing of using 4 cores with our parallel implementation using ThreadScope which indicates a very balanced load distribution.



Code (Haskell2048.hs):

```
{-# OPTIONS_GHC -Wall #-}
```

```
import Data.List     (elemIndices, intercalate, transpose, maximumBy)
import Data.Ord      (comparing)
import System.IO     (BufferMode(..), hSetBuffering, stdin)
import System.Random (randomRIO)
import Control.Parallel.Strategies

-- Each inner list is a row, starting with the top row
-- A 0 is an empty tile
type Board = [[Int]]


getMove :: Int -> Board -> Direction
-- parList rseq -- Parallel
-- rseq -- Sequential
getMove depth board = parMaximumOn (parList rseq) (score depth . flip slide board)
[minBound..maxBound]

score :: Int -> Board -> Int
score 0 board = length (emptyTiles board)
score depth board =
  case possibleNewTiles board of
    [] -> 0
    ts -> minimum (map (\b -> maximum (map (score (depth - 1) . flip slide b)
[minBound..maxBound] `using` parList rseq)) ts)

parMaximumOn :: (Ord b) => Strategy [(b,a)] -> (a -> b) -> [a] -> a
parMaximumOn strat f xs = snd (maximumBy (comparing fst) (map pairElementsWithScores xs
`using` strat))
  where
  pairElementsWithScores x = let w = f x in w `seq` (w, x)


-- ((,) $! f x) x

data Direction = North | East | South | West
  deriving (Enum, Bounded)

slideLeft :: Board -> Board
slideLeft = map slideRow
  where slideRow [ ] = [ ]
        slideRow [x] = [x]
        slideRow (x:y:zs)
          | x == 0 = slideRow (y : zs) ++ [0]
        | y == 0 = slideRow (x : zs) ++ [0] -- So that things will combine when 0's are between
them
        | x == y = (x + y) : slideRow zs ++ [0]
        | otherwise = x : slideRow (y : zs)

slide :: Direction -> Board -> Board
slide North = transpose . slideLeft . transpose
slide East  = map reverse . slideLeft . map reverse
slide South = transpose . map reverse . slideLeft . map reverse . transpose
slide West  = slideLeft

-- Tells us if the player won the game by getting a 2048 tile
completed :: Board -> Bool
completed b = any (elem 2048) b

-- Tells us if the game is over because there are no valid moves left
stalled :: Board -> Bool
stalled b = all stalled' b && all stalled' (transpose b)
  where stalled' row = notElem 0 row && noNeighbors row
        noNeighbors [ ] = True
        noNeighbors [_] = True
        noNeighbors (x:y:zs)
          | x == y    = False
          | otherwise = noNeighbors (y:zs)

-- Returns tuples of the indices of all of the empty tiles
emptyTiles :: Board -> [(Int, Int)]
emptyTiles = concatMap (uncurry search) . zip [0..3]
  where search n = zip (replicate 4 n) . elemIndices 0

-- Given a point, update replaces the value at the point on the board with the given value
updateTile :: (Int, Int) -> Int -> Board -> Board
updateTile (rowI, columnI) value = updateIndex (updateIndex (const value) columnI) rowI
  where updateIndex fn i list = take i list ++ fn (head $ drop i list) : tail (drop i list)

-- Adds a tile to a random empty spot.
-- 90% of the time the tile is a 2, 10% of the time it is a 4
addTile :: Board -> IO Board
addTile b = do
  let tiles = emptyTiles b
  newPoint <- randomRIO (0, length tiles - 1) >>= return . (tiles !!)
  newValue <- randomRIO (1, 10 :: Int) >>= return . \x -> if x == 1 then 4 else 2
  return $ updateTile newPoint newValue b
```

```haskell
possibleNewTiles :: Board -> [Board]
possibleNewTiles board = [ updateTile p v board | p <- emptyTiles board, v <- [2,4] ]

-- Our main game loop
gameloop :: Board -> IO ()
gameloop b = do
    putStrLn "---------------------"
    putStrLn $ boardToString b
    putStrLn "---------------------"
    if stalled b
        then putStrLn "Game over."
        else if completed b
            then putStrLn "You won!"
            else do
              let b1 = slide (getMove 2 b) b
              addTile b1 >>= gameloop

-- Board pretty printing
boardToString :: Board -> String
boardToString = init . unlines . map (vertical . map (pad . showSpecial))
  where vertical = ('|' :) . (++ "|") . intercalate "|"
        showSpecial 0 = ""
        showSpecial n = show n
        pad s = replicate (4 - length padTemp) ' ' ++ padTemp
          where padTemp = s ++ if length s < 3 then " " else ""

main :: IO ()
main = do
    hSetBuffering stdin NoBuffering
    let board = replicate 4 (replicate 4 0)
    b1 <- addTile board >>= addTile -- Add two tiles randomly and start the game!
    gameloop b1
```