COMS 4995 Parallel Functional Programming, Fall 2021

Final Project Report

Jeeho Song (js4892)

Parallel Minimax Agent : Implementation of Game 2048

Introduction

This project examines a Haskell parallel implementation of minimax algorithm in the game called 2048. The game 2048 is selected for the project because it is deterministic in a fully observable environment, and the minimax algorithm is complete and its running time can be adjusted with a max depth of the algorithm depending on the need of examination.

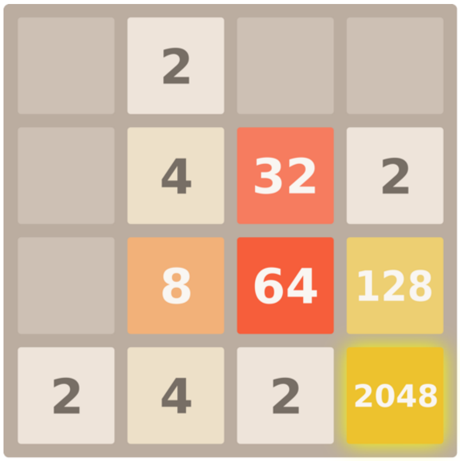
For the two-player turn-based game, the minimax algorithm is utilized to find an optimal choice for both agents. Particularly, the recurring process of minimax algorithm is dealt with parallel execution to speed up the running time of algorithm. For further improvement of algorithm, an iterative deepening search is used and also processed in a parallel programming. Although sophisticated heuristics methods might be a better way of improving the overall performance in this problem, I uses some naive heuristic methods that improve the program just as much as I can test the parallel implementation, in consideration of the purpose and the scope of the project.

Background

The game 2048 is a sliding tile puzzle game which was initially written in Javascript and CSS by Gabriele Circulli, who is an Italian web developer.[1] In a 4x4 grid, the game starts with two tiles with a value—which is either 2 or 4. Each player chooses a direction—up, down, left, and right—and then numbered tiles slide until they reach to another numbered tile or the edge of the grid. The two tiles are merged into one tile with a combined value when the two same numbered tiles are alongside in the direction of sliding. If three tiles of the same value are consecutive, then only the farthest two tiles along the sliding direction will be combined. If four tiles of the same value are consecutive, then the first two and the last two will be combined respectively. In each turn, a new value—either 2 (90%) or 4 (10%)—will be randomly set in a tile with 0 value. The goal of the game is to create a tile with a value of 2048. The total score, which starts from 0 and will be increased by a combined value when two tiles are merged, is kept track.

---

[1] https://github.com/gabrielecirulli/2048



Methodology

In this project, the program is designed to decide move direction on behalf of two players in each turn, and minimax algorithm is used for decisions. Minimax algorithm is a recursive or backtracking algorithm, based on an adversarial search, and mostly used in decision-making and game theory. The algorithm provides an optimal option for a player with an assumption that the opponent also chooses an optimal one in each state. Since the goal of each player is maximizing their own utility to win, both maximizer and minimizer functions are recursively used to find a decision in a state. The maximizer tries to get the highest value possible, while the minimizer tries to get the lowest score possible.

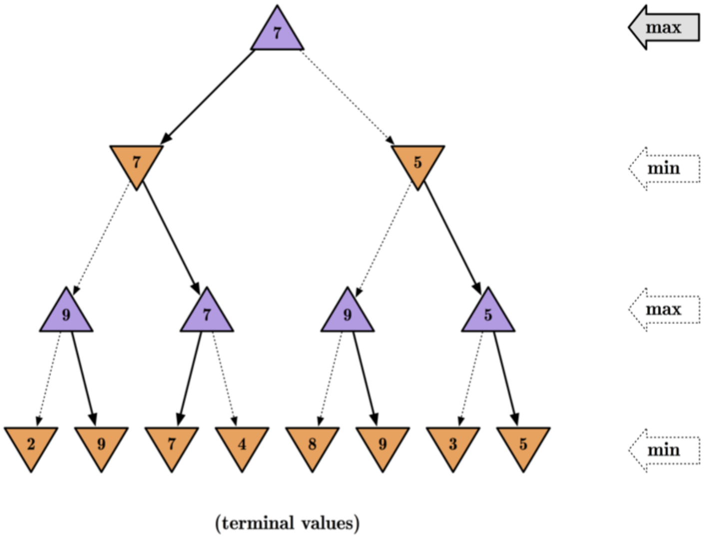


Figure 1 : a search tree

A search trees represents minimax algorithm that proceeds from the top node of the tree to the terminal node and backtracks the tree. Each node is an option containing a value that represents a utility, and a decision is made based on a comparison of two values in node pair. Terminal values are in nodes of the last level, the so-called leaf nodes. It is assumed that there are two players, MAX and MIN. In Figure 1, the MAX player chooses terminal values from leaf nodes to maximize the value to win the game. So, 9, 7, 9, 5 are selected. Then, the MIN player chooses those values to minimize the MAX's value to win, and 7, 5 are selected. In the same way, the MAX player chooses a bigger value to maximize. Finally, 7 is selected.

The value is the utility when the state is the leaf node. If the state is the MAX, the value is highest value of all successor node values, while the value is lowest value of all successor node values in the MIN.

The minimax algorithm can be represented in a pseudo-code as below:

```
Function minimax(node, isMaximizing):

    if depth == 0  or node is a terminal node then
        return the value of this node
    if isMaximizing
        for each child of node
            childValue = minimax(child, FALSE)
            value = max(value, childValue)
        return value
    else
        for each child of node
            childValue = minimax(child, TRUE)
            value = min(value, childValue)
        return value
```

Since the minimax algorithm loops through every node of the tree, it can be slow and inefficient depending on the depth. To improve the algorithm, we can use Alpha-beta pruning which decreases the number of nodes that minimax algorithm evaluates. Alpha is the largest value for MAX across evaluated nodes and beta is the lowest value for MIN. The initial alpha value is usually set as a negative infinity and the initial beta is set as a positive infinity, but in this project "minBound :: Int" and "maxBound :: Int" are respectively used because a possible value is bounded. As the algorithm goes over nodes, the values of alpha and beta are updated. And, when the alpha value is greater than the beta value, the remaining branches are pruned.

As noted above, some heuristic methods are used to increase the winning chance, which helps the analysis of parallel implementation. A couple of naive methods, which increase instantly the performance, are good enough for the scope of the project. A weight function is constructed with those methods; the goal value of 2048 has a large value of integer, a closer position to the top left corner gets a higher weight, tiles with a value of 0 have a weight, and neighboring tiles with similar values get higher weights.

Heuristic Methods

1) `getSpotZero :: Grid -> Integer`

```
   getSpotZero grid = toInteger $ sum $ map checkZeroSpot grid
```

```
        where checkZeroSpot grid = length $ filter isZero grid
                where isZero x = x == 0
```

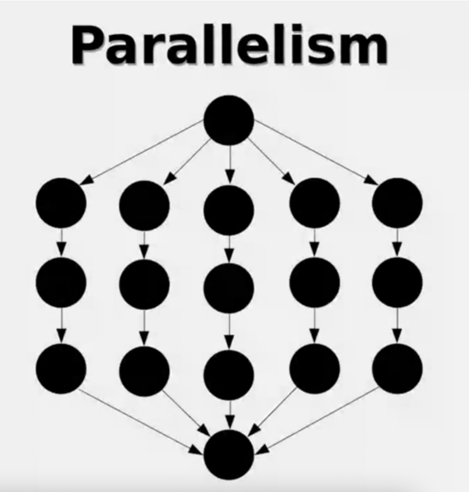
2) applyPositionWeights : new values of grid in cosideration of position heuristically

```
   applyPositionWeights :: Grid -> Integer
   applyPositionWeights grid = toInteger $ sum (map sum (zipWith (zipWith (*)) weights grid))
        where weights = [[10,8,6,4],[8,8,6,2],[6,6,4,1],[4,2,1,1]]
```

3) applySimilarityWeights :: Grid -> Integer

```
   applySimilarityWeights [] = 0
   applySimilarityWeights (x:xs) = smc x + applySimilarityWeights xs
        where smc (a:b:c:d:_) = fromIntegral $ 10*(abs (a - b) + abs (b - c) + abs (c - d))
                where smc [] = 0
```

In addition, different depths are applied to test the performance of the parallel execution. Since the time complexity of minimax is $O(b^d)$ and the minimax algorithm goes all the way down to the leaf node of the tree, which is not convenient for the purpose of examination, depths below 7 are tested given the running time constraints.



Parallelism

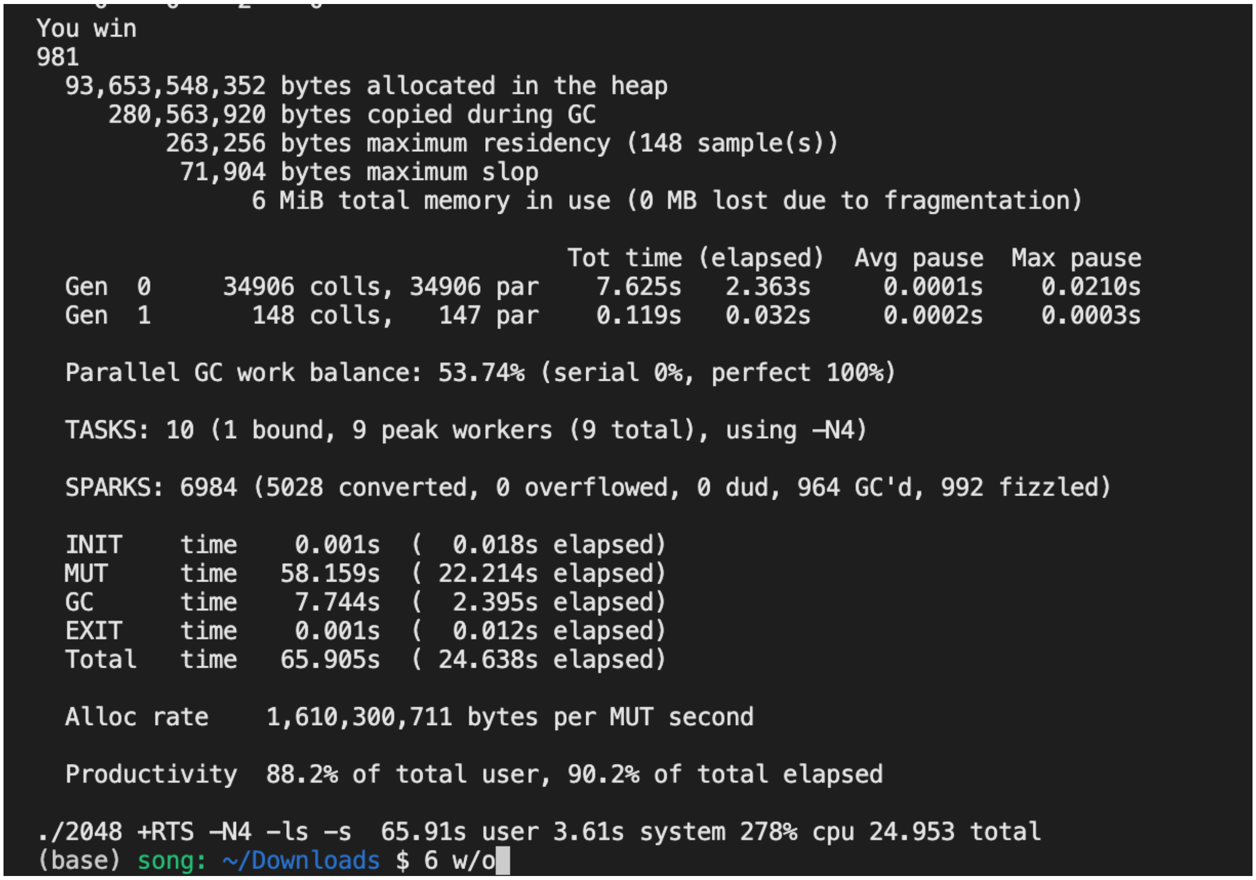
For a parallel execution of the algorithms in Haskell, Control.Parallel.Strategies is imported, and a few strategies in it are used. “rpar” is mainly used to spark the arguments, and “parList” evaluates the list of values, which is the derived utility. “runEval” is used to wrap out the result. “rseq” and “rdeepseq” are also tested respectively to examine performance difference. When the strategy is implemented, the program performs multiple executions for recursive search in parallel after being sparked and combines the results as a wrapped form.

The Strategies method is a deterministic parallel programming, and this project focuses only the method, but non-deterministic parallelism might be examined in future work. Haskell also provides a library called "Control.Concurrent" for that.

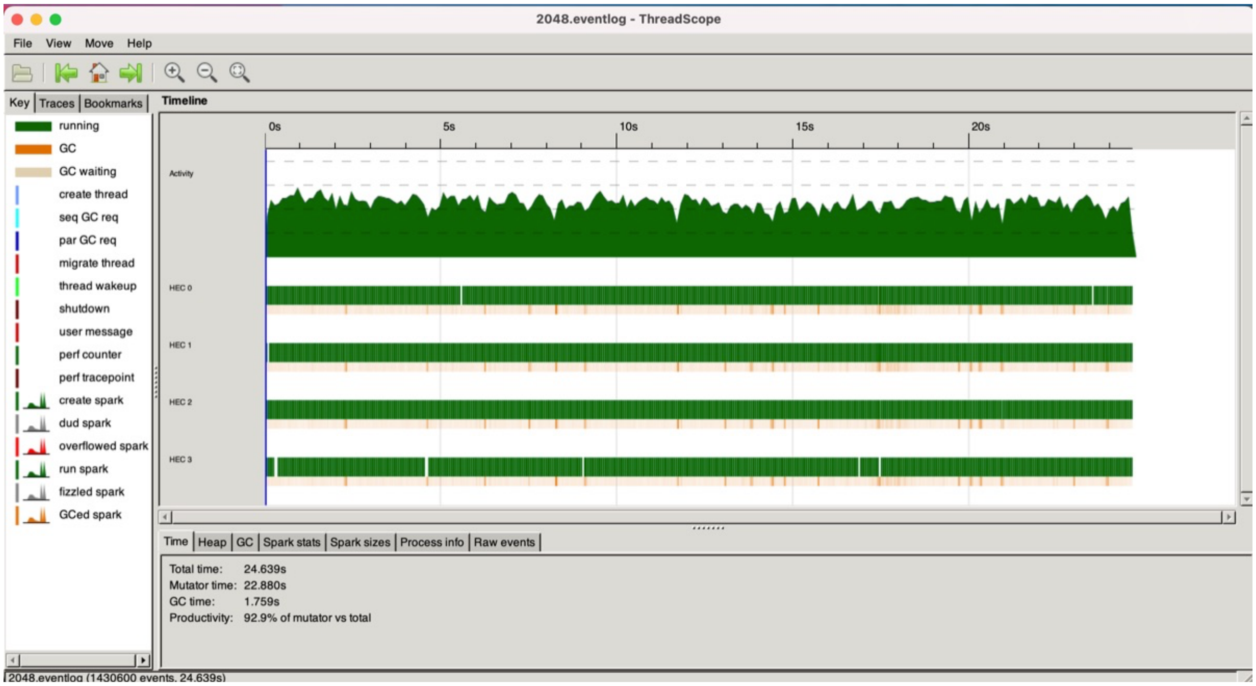
Result and Anlaysis

Since the minimax algorithm is involved in proceeding all the child nodes of each branch—by a depth, if any—in each move, it is understandable that the parallel programming reduces the total running time of the game. However, when Alpha-beta pruning is applied, I found that parallel implementation took longer than single-core sequential implementation.

Different depths are used for test, and the result of the parallel execution with the depth of 6 are displayed below.



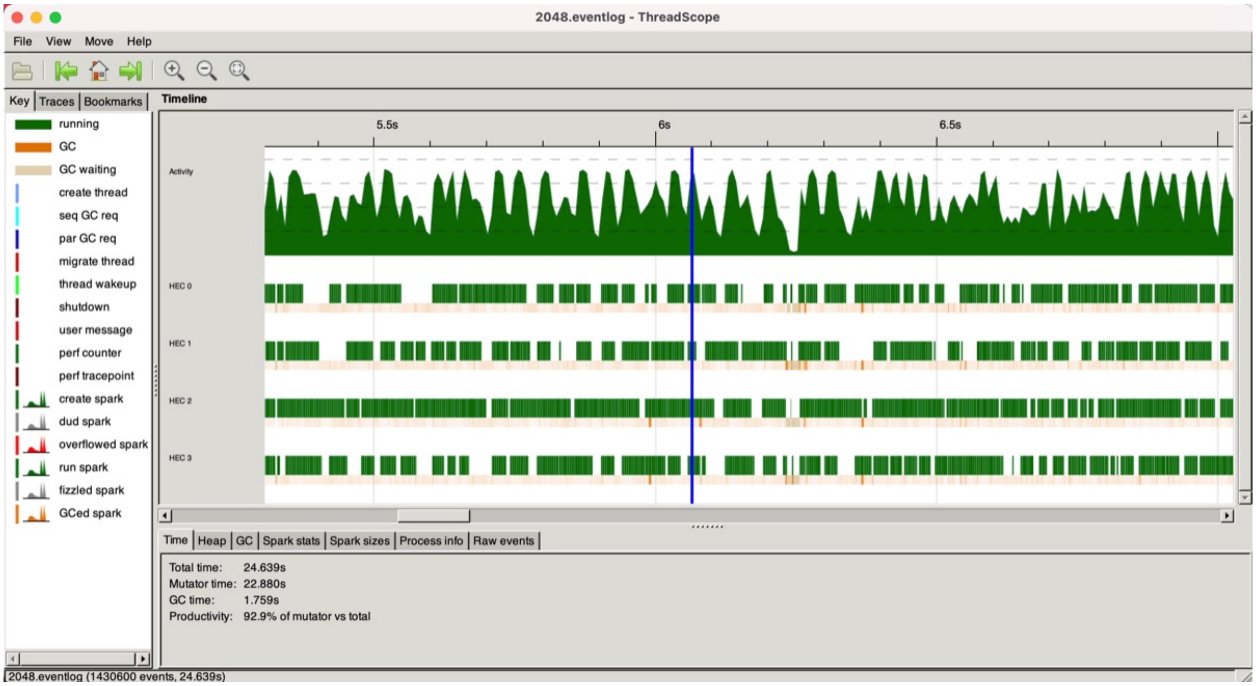
```
You win
981
  93,653,548,352 bytes allocated in the heap
     280,563,920 bytes copied during GC
         263,256 bytes maximum residency (148 sample(s))
          71,904 bytes maximum slop
               6 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0     34906 colls, 34906 par    7.625s   2.363s     0.0001s    0.0210s
  Gen  1       148 colls,   147 par    0.119s   0.032s     0.0002s    0.0003s

  Parallel GC work balance: 53.74% (serial 0%, perfect 100%)

  TASKS: 10 (1 bound, 9 peak workers (9 total), using -N4)

  SPARKS: 6984 (5028 converted, 0 overflowed, 0 dud, 964 GC'd, 992 fizzled)

  INIT    time    0.001s  (  0.018s elapsed)
  MUT     time   58.159s  ( 22.214s elapsed)
  GC      time    7.744s  (  2.395s elapsed)
  EXIT    time    0.001s  (  0.012s elapsed)
  Total   time   65.905s  ( 24.638s elapsed)

  Alloc rate    1,610,300,711 bytes per MUT second

  Productivity  88.2% of total user, 90.2% of total elapsed

./2048 +RTS -N4 -ls -s  65.91s user 3.61s system 278% cpu 24.953 total
(base) song: ~/Downloads $ 6 w/o
```



```
Total time:     24.639s
Mutator time:   22.880s
GC time:        1.759s
Productivity:   92.9% of mutator vs total
```

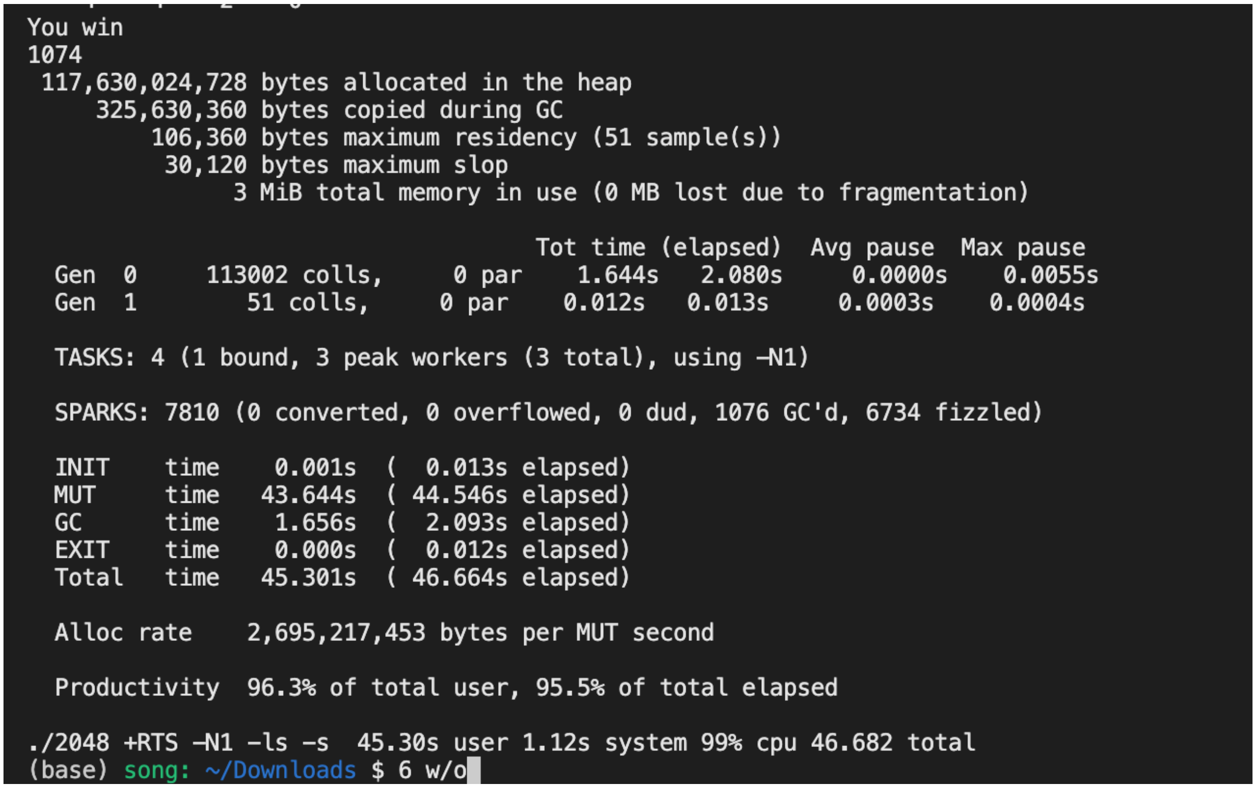
```
2048.eventlog (1430600 events, 24.639s)
```

In the 4-core processing, the total time is 65.905s, MUT time is 58.159s, and GC time is 7.744s. On the other hand, a reduced running time is observed in a single-core processing; the total time is 45.301s, MUT time is 43.644s, and GC time is 1.656s. Both MUT time and GC time are decreased. There is also a difference between 88.2% and 96.3% in Productivity. In the ThreadScope result, it looks like the 4 cores were all run regularly, but on a closer look (via zoom-in), it is not as follows:

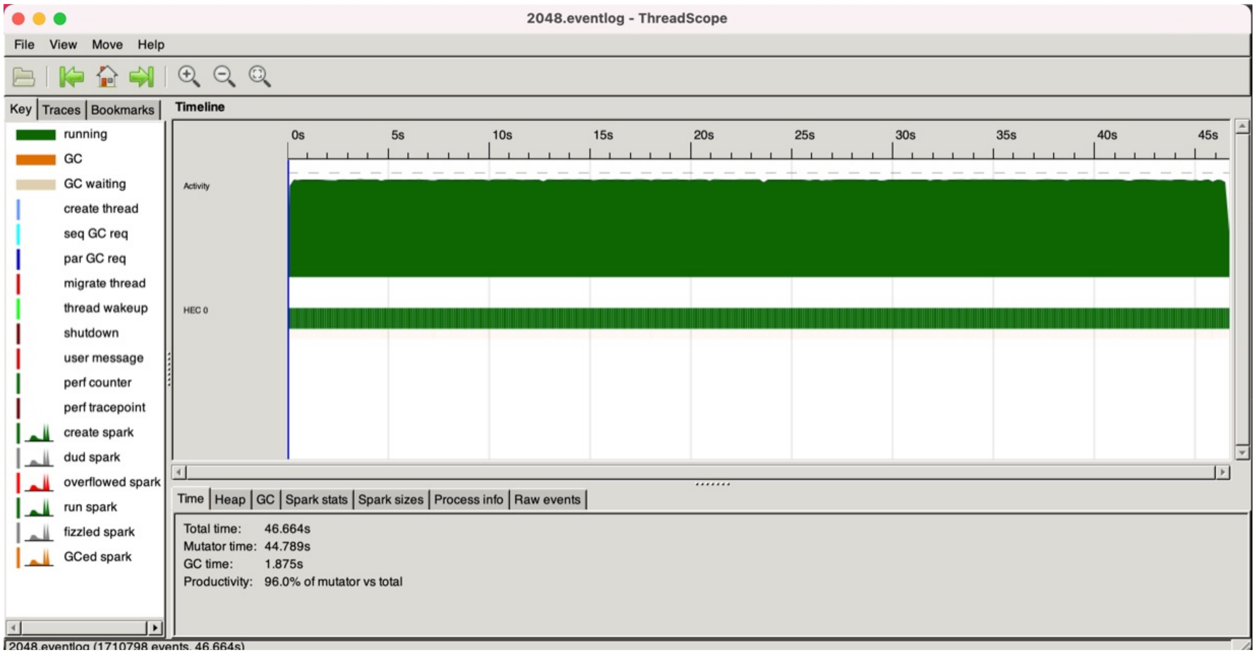


```
Total time:     24.639s
Mutator time:   22.880s
GC time:        1.759s
Productivity:   92.9% of mutator vs total
```

```
2048.eventlog (1430600 events, 24.639s)
```



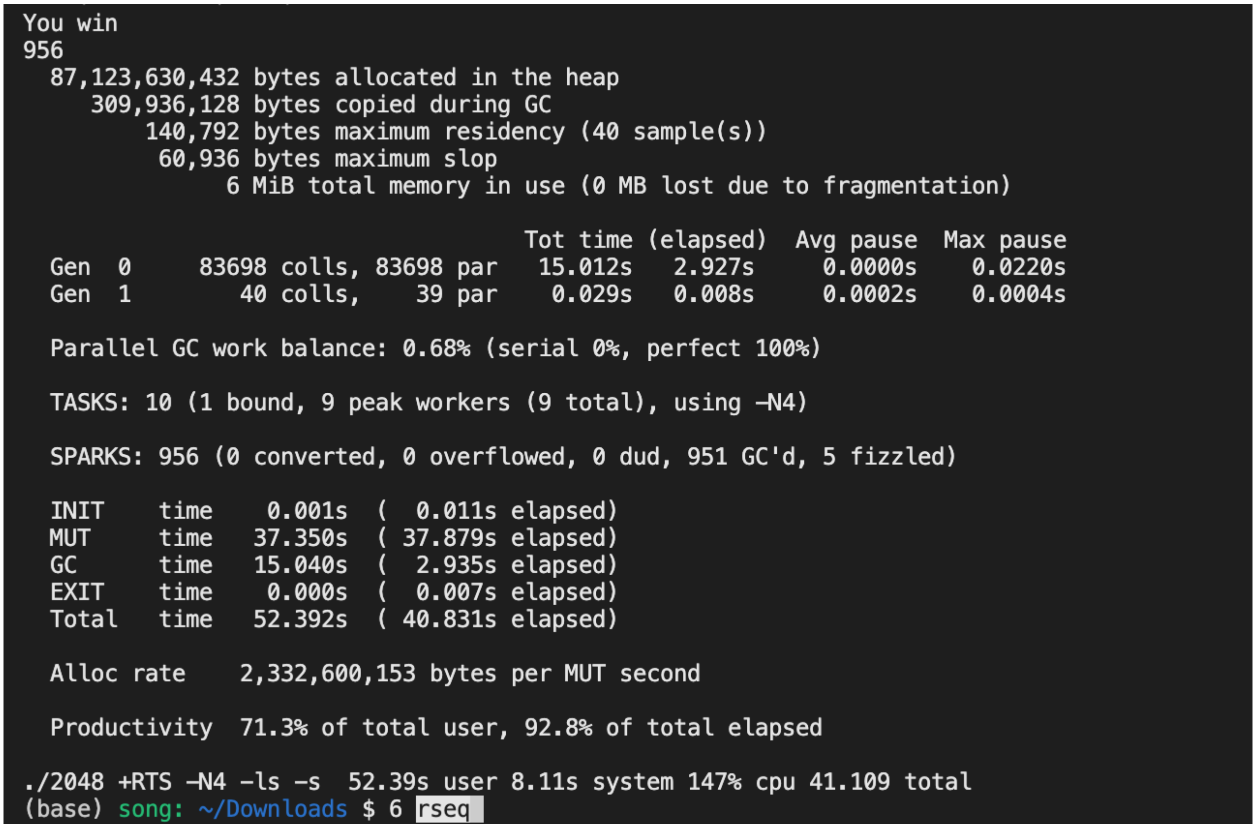
```
 You win
 1074
  117,630,024,728 bytes allocated in the heap
      325,630,360 bytes copied during GC
          106,360 bytes maximum residency (51 sample(s))
           30,120 bytes maximum slop
                3 MiB total memory in use (0 MB lost due to fragmentation)

                                      Tot time (elapsed)  Avg pause  Max pause
   Gen  0     113002 colls,     0 par    1.644s   2.080s     0.0000s    0.0055s
   Gen  1        51 colls,     0 par    0.012s   0.013s     0.0003s    0.0004s

   TASKS: 4 (1 bound, 3 peak workers (3 total), using -N1)

   SPARKS: 7810 (0 converted, 0 overflowed, 0 dud, 1076 GC'd, 6734 fizzled)

   INIT    time    0.001s  (  0.013s elapsed)
   MUT     time   43.644s  ( 44.546s elapsed)
   GC      time    1.656s  (  2.093s elapsed)
   EXIT    time    0.000s  (  0.012s elapsed)
   Total   time   45.301s  ( 46.664s elapsed)

   Alloc rate    2,695,217,453 bytes per MUT second

   Productivity  96.3% of total user, 95.5% of total elapsed

 ./2048 +RTS -N1 -ls -s  45.30s user 1.12s system 99% cpu 46.682 total
 (base) song: ~/Downloads $ 6 w/o
```



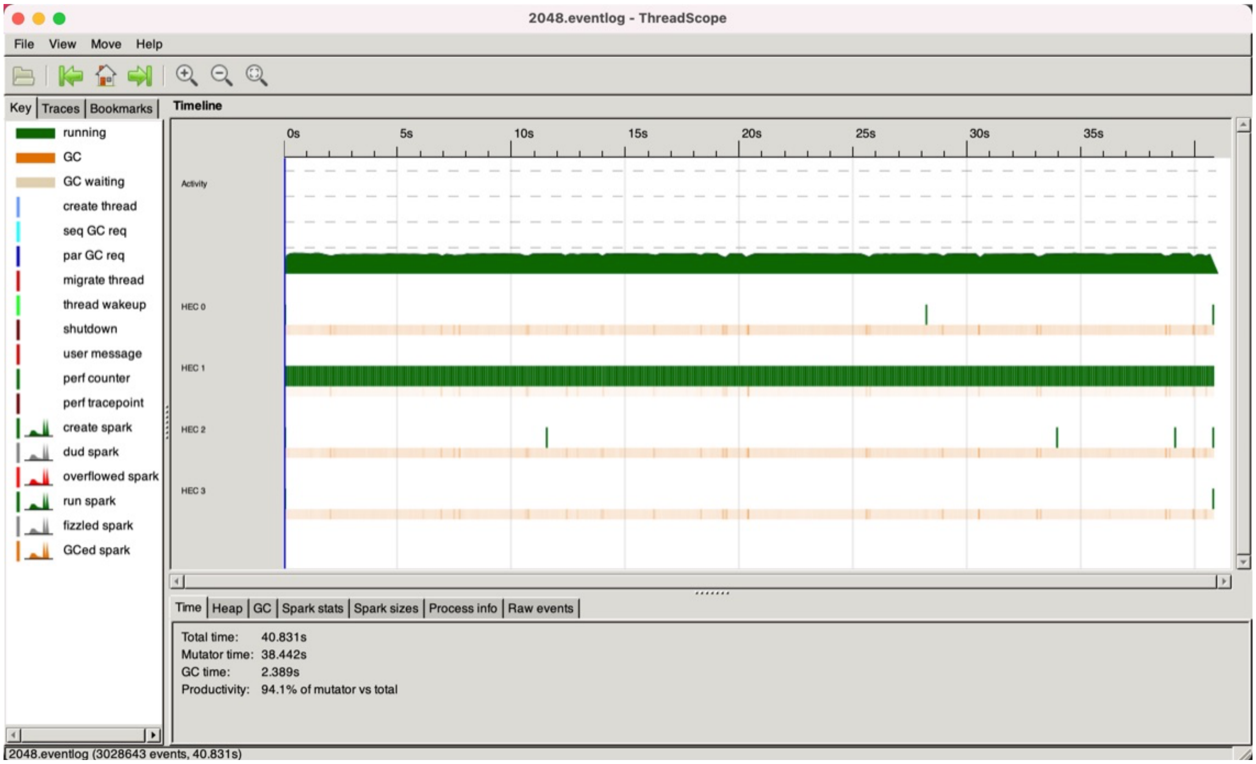
```
Total time:     46.664s
Mutator time:   44.789s
GC time:        1.875s
Productivity:   96.0% of mutator vs total
```

Much more regular and active processing is monitored in a single-core processing.

For further examination, “rseq” is added, and the result is as follows:

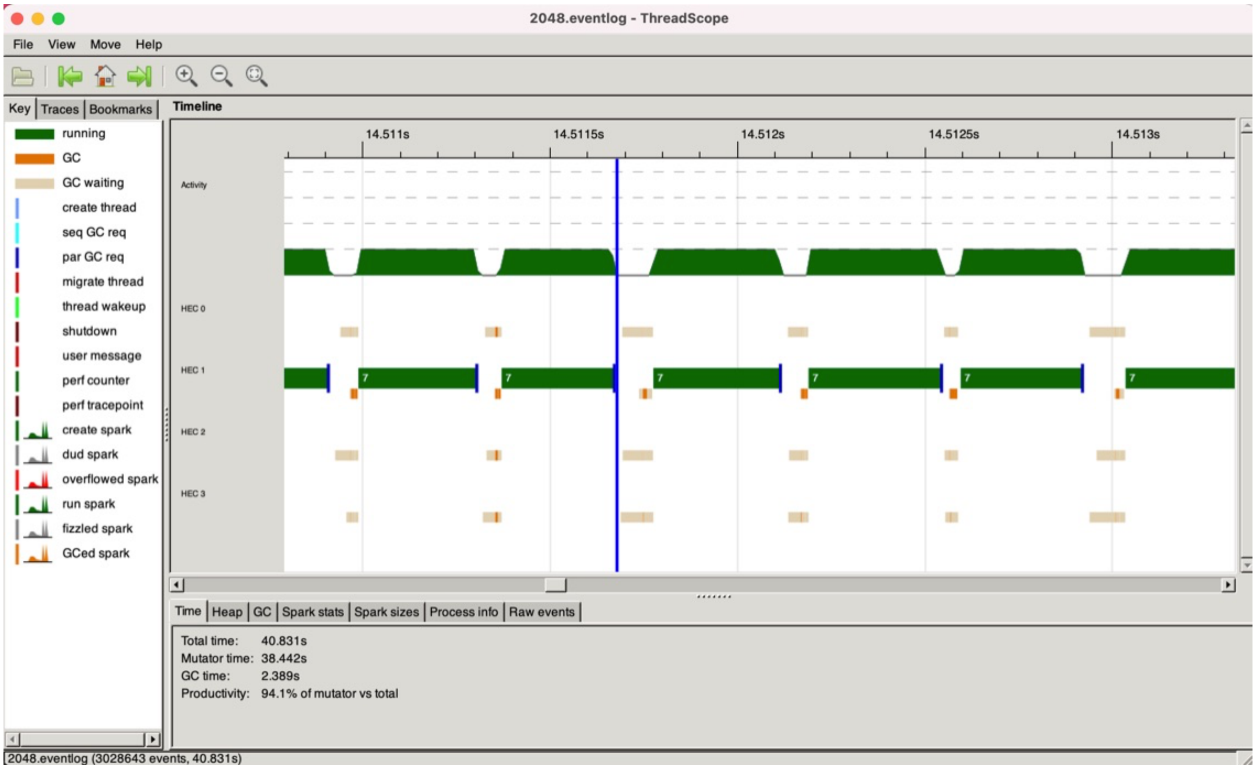


```
You win
956
  87,123,630,432 bytes allocated in the heap
     309,936,128 bytes copied during GC
         140,792 bytes maximum residency (40 sample(s))
          60,936 bytes maximum slop
               6 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0     83698 colls, 83698 par   15.012s   2.927s     0.0000s    0.0220s
  Gen  1        40 colls,    39 par    0.029s   0.008s     0.0002s    0.0004s

  Parallel GC work balance: 0.68% (serial 0%, perfect 100%)

  TASKS: 10 (1 bound, 9 peak workers (9 total), using -N4)

  SPARKS: 956 (0 converted, 0 overflowed, 0 dud, 951 GC'd, 5 fizzled)

  INIT    time    0.001s  (  0.011s elapsed)
  MUT     time   37.350s  ( 37.879s elapsed)
  GC      time   15.040s  (  2.935s elapsed)
  EXIT    time    0.000s  (  0.007s elapsed)
  Total   time   52.392s  ( 40.831s elapsed)

  Alloc rate    2,332,600,153 bytes per MUT second

  Productivity  71.3% of total user, 92.8% of total elapsed

./2048 +RTS -N4 -ls -s  52.39s user 8.11s system 147% cpu 41.109 total
(base) song: ~/Downloads $ 6 rseq
```



Total time: 40.831s
Mutator time: 38.442s
GC time: 2.389s
Productivity: 94.1% of mutator vs total

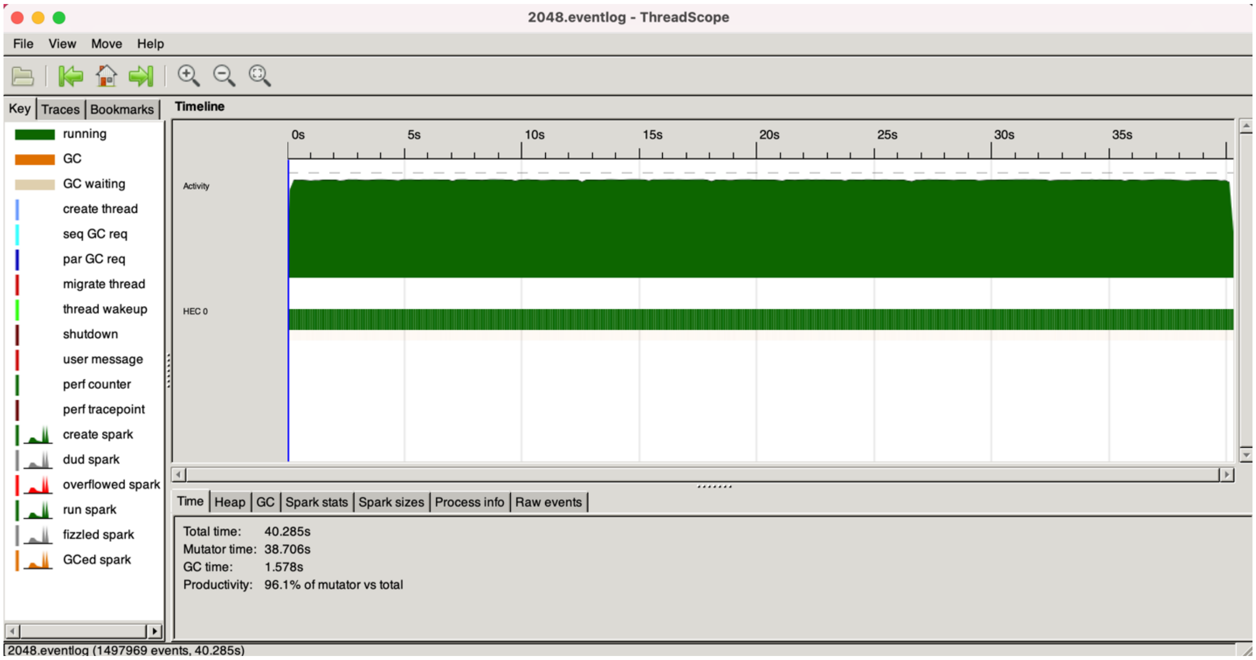
2048.eventlog (3028643 events, 40.831s)



```
Total time:     40.831s
Mutator time:   38.442s
GC time:        2.389s
Productivity:   94.1% of mutator vs total
```

```
2048.eventlog (3028643 events, 40.831s)
```

The number of sparks is significantly reduced in this rseq application. In a 4-core processing, there is no converted spark, and the three cores did not work properly.

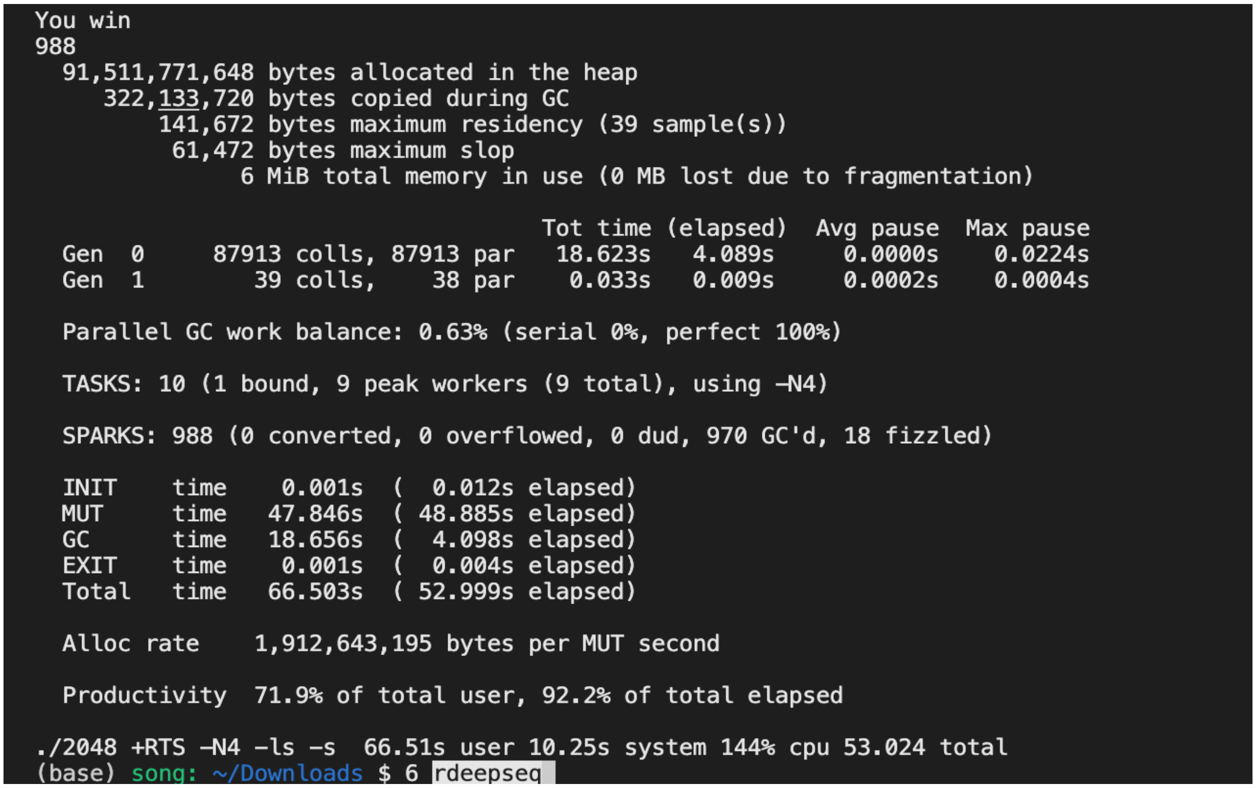


```
Total time:     40.285s
Mutator time:   38.706s
GC time:        1.578s
Productivity:   96.1% of mutator vs total
```

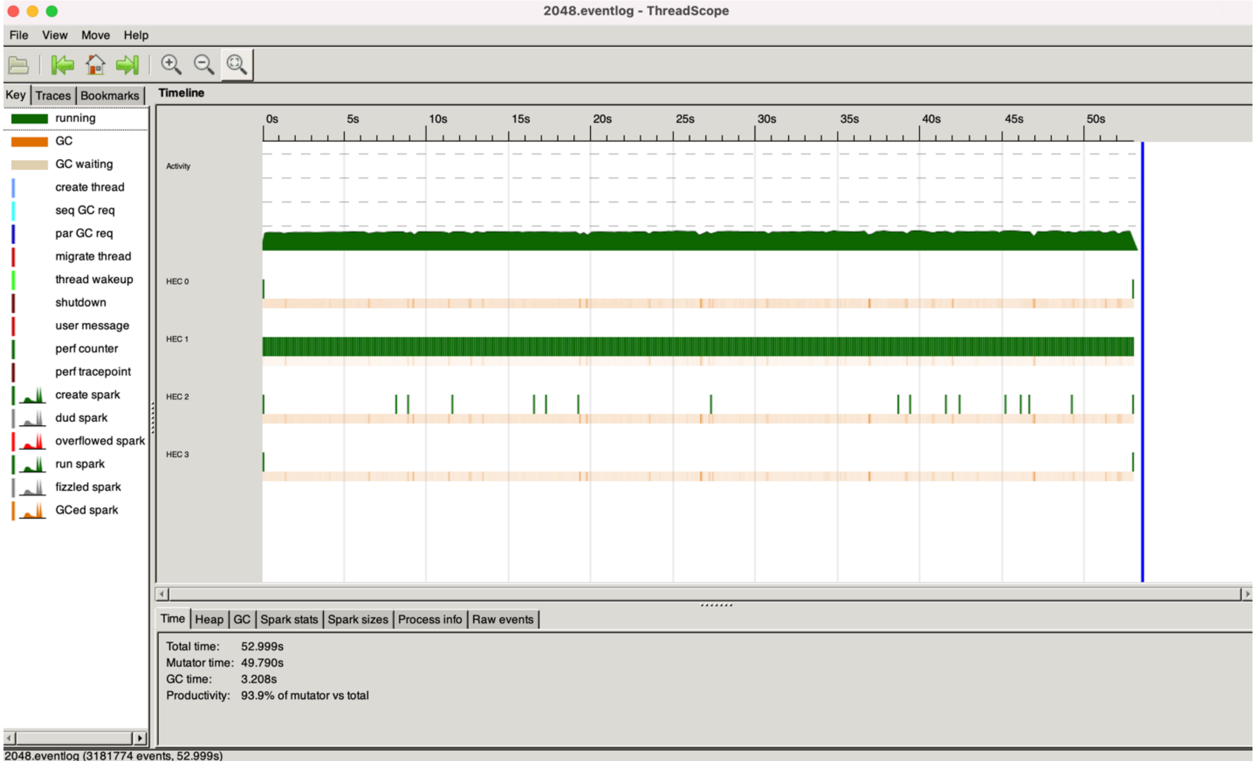
```
2048.eventlog (1497969 events, 40.285s)
```

On the other hand, a single-core processing worked well without interruption.

It is much clear in using rdeepseq to evaluate the argument. “rdeepseq” is slower, but it increases considerably the chance of winning.

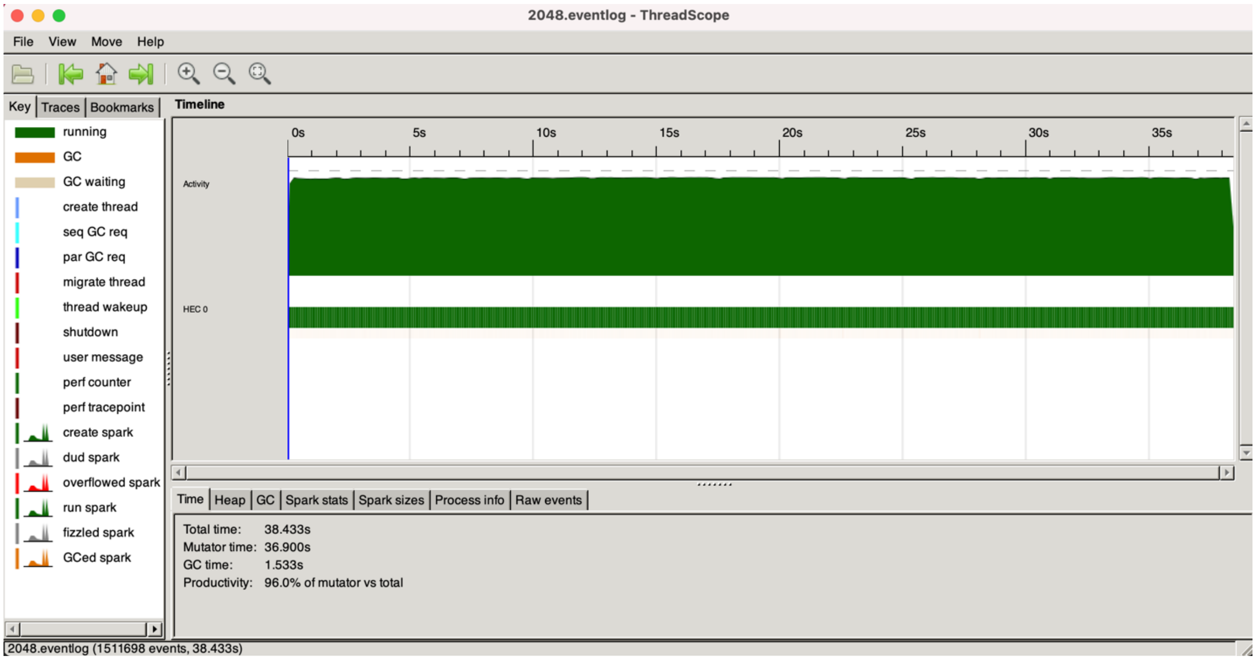


```
 You win
 988
   91,511,771,648 bytes allocated in the heap
      322,133,720 bytes copied during GC
          141,672 bytes maximum residency (39 sample(s))
           61,472 bytes maximum slop
                6 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
   Gen  0     87913 colls, 87913 par   18.623s   4.089s     0.0000s    0.0224s
   Gen  1        39 colls,    38 par    0.033s   0.009s     0.0002s    0.0004s

   Parallel GC work balance: 0.63% (serial 0%, perfect 100%)

   TASKS: 10 (1 bound, 9 peak workers (9 total), using -N4)

   SPARKS: 988 (0 converted, 0 overflowed, 0 dud, 970 GC'd, 18 fizzled)

   INIT    time    0.001s  (  0.012s elapsed)
   MUT     time   47.846s  ( 48.885s elapsed)
   GC      time   18.656s  (  4.098s elapsed)
   EXIT    time    0.001s  (  0.004s elapsed)
   Total   time   66.503s  ( 52.999s elapsed)

   Alloc rate    1,912,643,195 bytes per MUT second

   Productivity  71.9% of total user, 92.2% of total elapsed

 ./2048 +RTS -N4 -ls -s  66.51s user 10.25s system 144% cpu 53.024 total
 (base) song: ~/Downloads $ 6 rdeepseq
```



Total time: 52.999s
Mutator time: 49.790s
GC time: 3.208s
Productivity: 93.9% of mutator vs total

2048.eventlog (3181774 events, 52.999s)



```
Total time:     38.433s
Mutator time:   36.900s
GC time:        1.533s
Productivity:   96.0% of mutator vs total
```

```
2048.eventlog (1511698 events, 38.433s)
```

Conclusion

The parallel programming is indeed impactful for the performance of the program. If it is properly used in a certain algorithm, it can increase the speed of run-time and thus efficiency. However, an actual parallel execution under the hood is abstract. As examined above, the parallel implementation increased the performance of the game by reducing the running time, but it did not work as expected when the alpha-beta pruning method is applied. In addition, “rdeepseq” is slower, but it seems to increase the accuracy of minimax.

Reference

In consideration of the scope of the project, an existing code[2] of the game 2048 in GitHub is partly used.

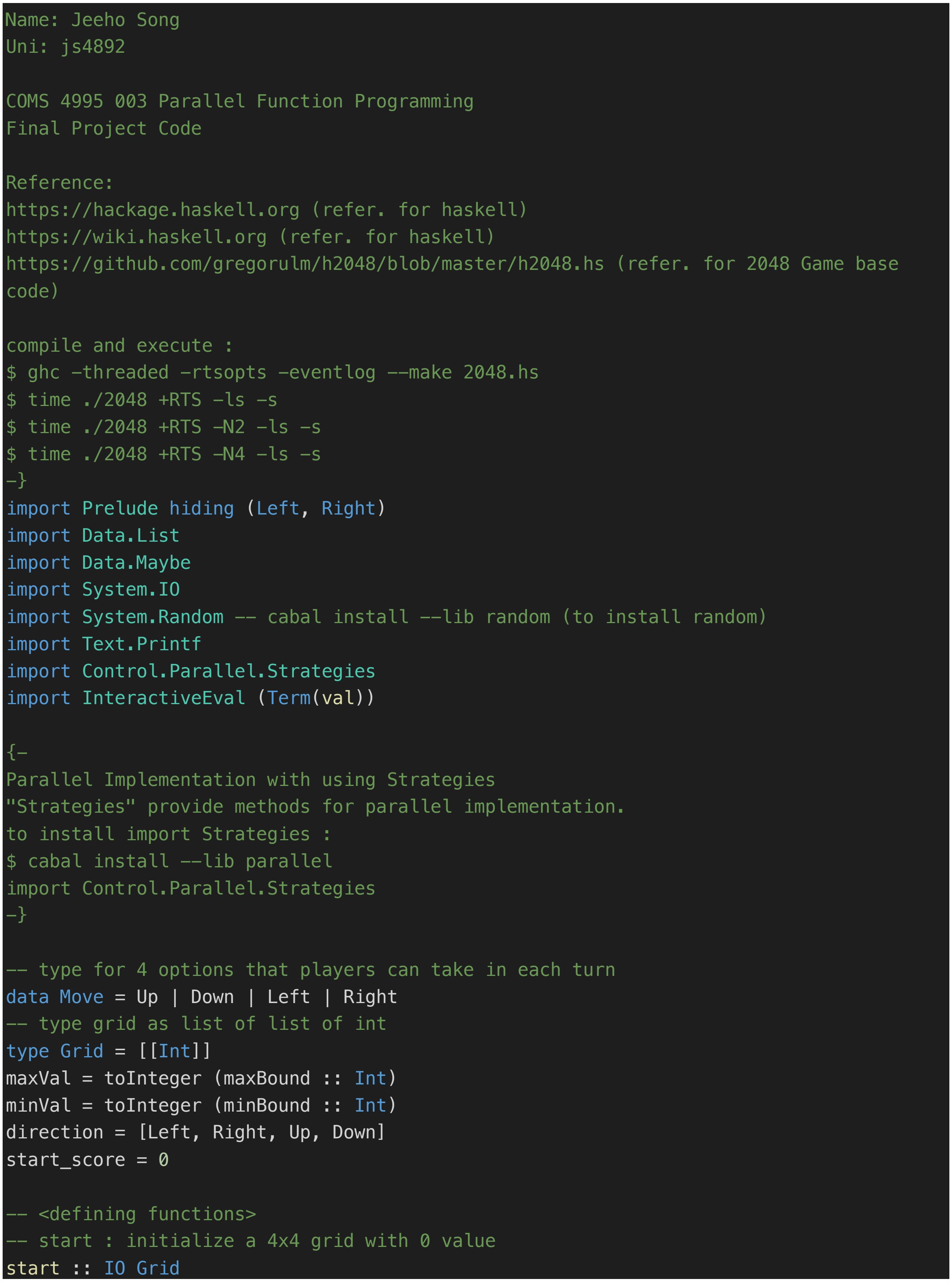
https://github.com/gregorulm/h2048/blob/master/h2048.hs
https://hackage.haskell.org (refer. for haskell)
https://wiki.haskell.org (refer. for haskell)

Code

```
{-
```

---

[2] https://github.com/gregorulm/h2048

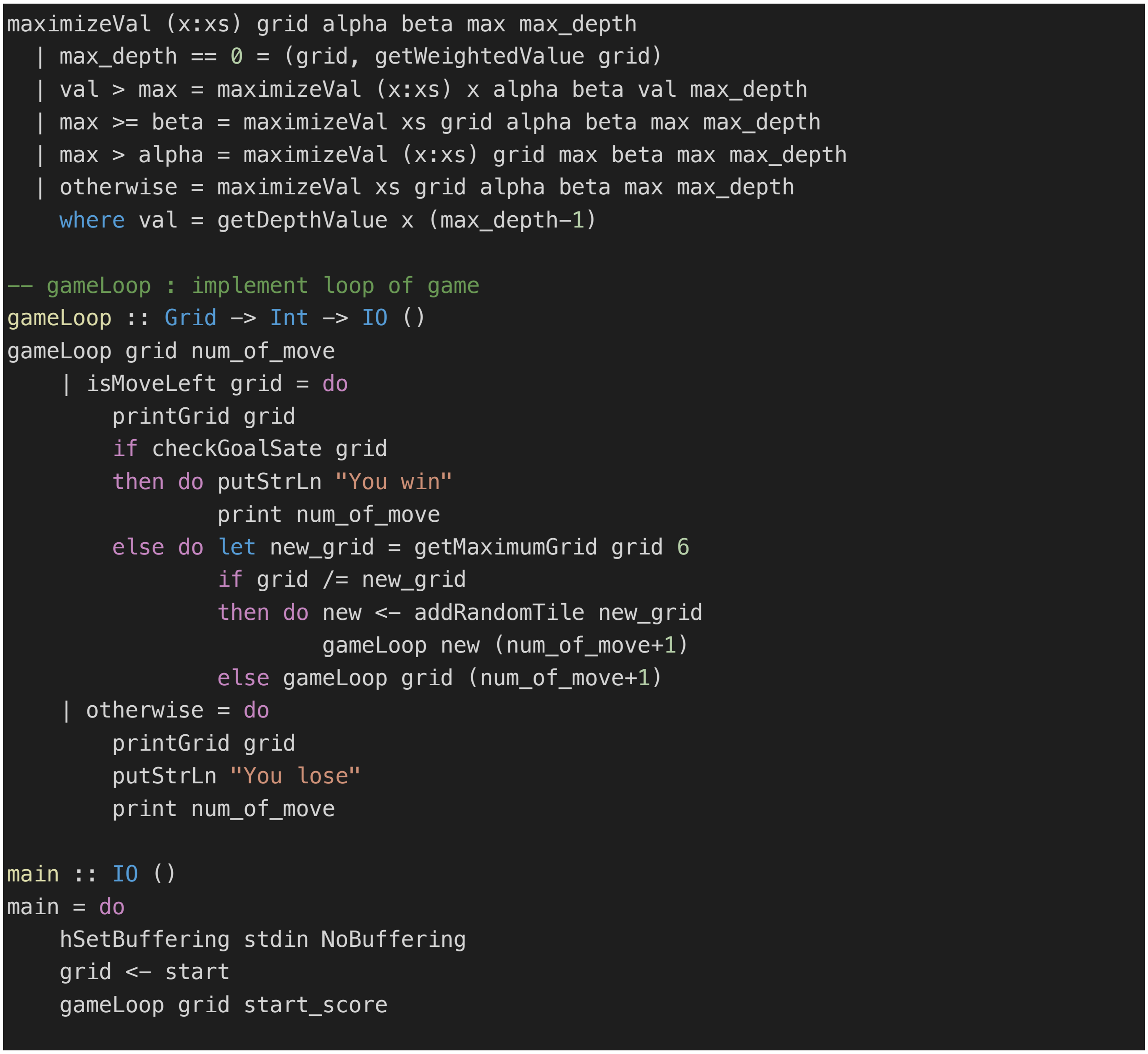


```
Name: Jeeho Song
Uni: js4892

COMS 4995 003 Parallel Function Programming
Final Project Code

Reference:
https://hackage.haskell.org (refer. for haskell)
https://wiki.haskell.org (refer. for haskell)
https://github.com/gregorulm/h2048/blob/master/h2048.hs (refer. for 2048 Game base
code)

compile and execute :
$ ghc -threaded -rtsopts -eventlog --make 2048.hs
$ time ./2048 +RTS -ls -s
$ time ./2048 +RTS -N2 -ls -s
$ time ./2048 +RTS -N4 -ls -s
-}
import Prelude hiding (Left, Right)
import Data.List
import Data.Maybe
import System.IO
import System.Random -- cabal install --lib random (to install random)
import Text.Printf
import Control.Parallel.Strategies
import InteractiveEval (Term(val))

{-
Parallel Implementation with using Strategies
"Strategies" provide methods for parallel implementation.
to install import Strategies :
$ cabal install --lib parallel
import Control.Parallel.Strategies
-}

-- type for 4 options that players can take in each turn
data Move = Up | Down | Left | Right
-- type grid as list of list of int
type Grid = [[Int]]
maxVal = toInteger (maxBound :: Int)
minVal = toInteger (minBound :: Int)
direction = [Left, Right, Up, Down]
start_score = 0

-- <defining functions>
-- start : initialize a 4x4 grid with 0 value
start :: IO Grid
```

```haskell
start = do grid' <- addRandomTile $ replicate 4 [0, 0, 0, 0]
           addRandomTile grid'

-- addRandomTile : add a value of 2 (90%) or 4(10%) in a tile with 0 value
addRandomTile :: Grid -> IO Grid
addRandomTile grid = do
    val <- pickRandomEmptySpot [2,2,2,2,2,2,2,2,2,4]
    let randomSpot = getCoordinateOfZero grid
    selected_random_spot <- pickRandomEmptySpot randomSpot
    let new_grid = setNewGrid grid selected_random_spot val
    return new_grid

-- getCoordinateOfZero : get coordinates of tiles with 0 value
getCoordinateOfZero :: Grid -> [(Int, Int)]
getCoordinateOfZero grid = filter (\(row, col) -> (grid!!row)!!col == 0) coordinates
    where singleRow n = zip (replicate 4 n) [0..3]
          coordinates = concatMap singleRow [0..3]

-- pickRandomEmptySpot : choose randomly where a new tile with 2 or 4 is set among an empty tile
pickRandomEmptySpot :: [a] -> IO a
pickRandomEmptySpot xs = do
    i <- randomRIO (0, length xs-1)
    return (xs !! i)

-- setNewGrid : set a new grid configuration
setNewGrid :: Grid -> (Int, Int) -> Int -> Grid
setNewGrid grid (row, col) val = pre ++ [mid] ++ post
    where pre  = take row grid
          mid  = take col (grid!!row) ++ [val] ++ drop (col + 1) (grid!!row)
          post = drop (row + 1) grid

-- merge : combine two same numbered tiles
merge :: [Int] -> [Int]
merge xs = merged ++ padding
    where padding = replicate (length xs - length merged) 0
          merged  = combine $ filter (/= 0) xs
          combine (x:y:ys) | x == y = x * 2 : combine ys
                           | otherwise = x  : combine (y:ys)
          combine x = x

-- move : get a new grid after move with direction (up, down, left, right)
move :: Move -> Grid -> Grid
move Left  = map merge
move Right = map (reverse . merge . reverse)
move Up    = transpose . move Left  . transpose
move Down  = transpose . move Right . transpose
```

```haskell
-- isMoveLeft : check if it is applied in "LEFT"
isMoveLeft :: Grid -> Bool
isMoveLeft grid = sum allChoices > 0
    where allChoices = map (length . getCoordinateOfZero . flip move grid) directions
          directions = [Left, Right, Up, Down]

-- printGrid : display the current grid
printGrid :: Grid -> IO ()
printGrid grid = do
    putStr "\n"
    mapM_ (putStrLn . concatMap (printf "%5d")) grid

-- checkGoalSate : check if it is a goal state (a tile with the value of 2048)
checkGoalSate :: Grid -> Bool
checkGoalSate grid = [] /= filter (== 2048) (concat grid)

-- get child nodes grid array
getChildNodes :: Grid -> [Grid]
getChildNodes grid = filter isNotGrid [move direction grid | direction <- direction ]
    where isNotGrid values = values /= grid

-- goal state incentive + 3 hueristic methods
-- getWeightedValue : get weighted value (after applying heuristic methods)
getWeightedValue :: Grid -> Integer
getWeightedValue grid = maximum [if x == 2048 then 1000 else 0 | x <- map maximum grid] + applyPositionWeights grid - applySimilarityWeights grid + (100*getSpotZero grid)

-- 3 Heuristic Methods
getSpotZero :: Grid -> Integer
getSpotZero grid = toInteger $ sum $ map checkZeroSpot grid
    where checkZeroSpot grid = length $ filter isZero grid
            where isZero x = x == 0

-- applyPositionWeights : new values of grid in cosideration of position heuristically
applyPositionWeights :: Grid -> Integer
applyPositionWeights grid = toInteger $ sum (map sum (zipWith (zipWith (*)) weights grid))
    where weights = [[10,8,6,4],[8,8,6,2],[6,6,4,1],[4,2,1,1]]

-- applySimilarityWeights : get tiles with similar values closer
applySimilarityWeights :: Grid -> Integer
applySimilarityWeights [] = 0
applySimilarityWeights (x:xs) = smc x + applySimilarityWeights xs
    where smc (a:b:c:d:_) = fromIntegral $ 10*(abs(a-b) + abs(b-c) + abs(c-d))
            where smc [] = 0
```

```haskell
-- parallel implementation for minimax (using Strategies)
-- getMaximumGrid : find a next grid that maximizes the winning chance in paralell implementation
getMaximumGrid :: Grid -> Int -> Grid
getMaximumGrid grid max_depth = new_grid !! fromJust (elemIndex (maximum values) values)
    where new_grid = getChildNodes grid
          values = runEval $ do {
              ; result <- rpar [getDepthValue grid (max_depth-1) | grid <- new_grid]
              ; rseq result
            --   ; rdeepseq result
              ; return result
          }
    -- runEval :: Eval a -> a (to pull the result out of the monad)
    -- parList :: Strategy a -> Strategy [a] (to evaluate each element of a list in parallel according to given strategy)
    -- rpar :: Strategy a (to spark its argument (for evaluation in parallel))
    -- rseq :: Strategy a (to evaluates its argument to weak head normal form)
    -- rdeepseq :: NFData a => Strategy a (to fully evaluates its argument)

-- getDepthValue : get values of nodes in the depth
getDepthValue :: Grid -> Int -> Integer
getDepthValue grid 0 = getWeightedValue grid
getDepthValue grid depth = toInteger twoVal + toInteger fourVal
    where twoVal = round(realToFrac(minimizeVal grid (getCoordinateOfZero grid) 2 minVal maxVal maxVal (depth-1))*0.9)
          fourVal = round(realToFrac(minimizeVal grid (getCoordinateOfZero grid) 4 minVal maxVal maxVal (depth-1))*0.9)

-- minimax algorithm : minimizeVal & maxmizeVal
minimizeVal :: Grid -> [(Int, Int)] -> Int -> Integer -> Integer -> Integer -> Int -> Integer
minimizeVal grid [] value alpha beta min depth = min
minimizeVal grid (x:xs) value alpha beta min depth
  | depth == 0 = getWeightedValue grid
  | val < min = minimizeVal grid (x:xs) value alpha beta val depth
  | beta > min = minimizeVal grid (x:xs) value alpha min min depth
  | alpha >= min = minimizeVal grid xs value alpha beta min depth
  | otherwise = minimizeVal grid xs value alpha beta min depth
    where (netSet, val) = maximizeVal (getChildNodes (setNewGrid grid x value)) grid alpha beta minVal (depth-1)

maximizeVal :: [Grid] -> Grid -> Integer -> Integer -> Integer -> Int -> (Grid, Integer)
maximizeVal [] grid alpha beta max max_depth = (grid, max)
```



```haskell
maximizeVal (x:xs) grid alpha beta max max_depth
  | max_depth == 0 = (grid, getWeightedValue grid)
  | val > max = maximizeVal (x:xs) x alpha beta val max_depth
  | max >= beta = maximizeVal xs grid alpha beta max max_depth
  | max > alpha = maximizeVal (x:xs) grid max beta max max_depth
  | otherwise = maximizeVal xs grid alpha beta max max_depth
    where val = getDepthValue x (max_depth-1)

-- gameLoop : implement loop of game
gameLoop :: Grid -> Int -> IO ()
gameLoop grid num_of_move
    | isMoveLeft grid = do
        printGrid grid
        if checkGoalSate grid
        then do putStrLn "You win"
                print num_of_move
        else do let new_grid = getMaximumGrid grid 6
                if grid /= new_grid
                then do new <- addRandomTile new_grid
                        gameLoop new (num_of_move+1)
                else gameLoop grid (num_of_move+1)
    | otherwise = do
        printGrid grid
        putStrLn "You lose"
        print num_of_move

main :: IO ()
main = do
    hSetBuffering stdin NoBuffering
    grid <- start
    gameLoop grid start_score
```