1 Introduction
This paper presents a parallelized maze generator implemented in Haskell. There are several algorithms to generate a maze, such as depth-first search, Kruskal’s algorithm, Prim’s algorithm. However, most of the algorithms are hard to run in parallel. In order to demonstrate the parallelization techniques in Haskell, we decided to use another algorithm, which is known as the Recursive Division algorithm. It is the fastest maze generation algorithm without directional biases thanks to its parallelism, each sub-maze could be processed at finer and finer levels of detail. However, compared to other more convoluted maze generation algorithms, the output maze usually contains long straight walls crossing the space and looks less complicated.

2 Recursive Division Algorithm
The Recursive Division works as follows:
1. Initialize the maze with no walls inside, i.e. only a frame.

2. Divide the frame into four separate spaces by adding two randomly positioned walls, one vertically and one horizontally.
3. Randomly selected three from the four spaces and connected them together by opening three randomly positioned passages on the walls.

4. Recursively repeat the process on the sub maze until all sub maze can’t be divided anymore.

2.1 Data Types

Maze could be abstracted as a grid, each passage in the maze could be seen as a cell in the grid, but there are usually two ways to represent the walls in the maze. One is using lines in the grid to represent the walls, whereas the other way represents walls also as cells. In our implementation, we represent both passage and wall using the cells.
We defined two main data types for our implementation, Maze and Wall. The Maze type is the abstract of the generated maze, with the property width and height represented by m and n. It also contains a set of Wall types, representing the walls in the maze.

```hs
data Maze = Maze {
  m :: Int, -- number of matrix rows
  n :: Int, -- number of matrix cols
  walls :: Set W.Wall
}

instance Show Maze where
  show (Maze m' n' walls') =
    unlines maze
    where
      maze = [rowToStrng r| r <- [0 .. m'-1]]
        rowToStrng r = [if isWall r c then '#' else ' '| c <- [0 .. n'-1]]
        isWall r c = W.Wall r c `elem` walls'

instance NFData Maze where
  rnf (Maze m' n' walls') = rnf m' `seq` rnf n' `seq` rnf walls'
```
The data type Wall has the properties r and c representing the row and column number of this wall block respectively.

```haskell
data Wall = Wall {
    r :: Int,
    c :: Int
} deriving (Show, Eq, Ord)

instance NFData Wall where
    rnf (Wall r c) = rnf r `seq` rnf c
```

## 2.2 Implementation

The entry point of the algorithm is the function `generateWalls`, which generates walls in the area defined by four integers (x0, y0), (x1, y1) representing the top left and bottom right corners.

In this function, we split the given area into four sub mazes and solved each of them individually. In the end, we return the result by combining the four solutions.

```haskell
-- Given an frame, generate walls inside the frame
-- top left wall cell: (tr, tc)
-- bottom right wall cell: (br, bc)
generateWalls :: RandomGen g => Int -> Int -> Int -> Int -> g -> Set W.Wall
generateWalls tr tc br bc g |
    | bc - tc < 4 || br - tr < 4 = empty |
    | otherwise = walls
    | 'union` topLeft
    | 'union` topRight
    | 'union` bottomLeft
    | 'union` bottomRight
where
    (g1, g2) = split g
    (g3, g4) = split g1
    (g5, g6) = split g2
    (g7, g8) = split g3
    (randomRow, _) = pickRandom [(tr + 2), (tr + 4)..(br - 2)]
    g1 (randomCol, _) = pickRandom [(tc + 2), (tc + 4)..(bc - 2)]
    g2 walls = verticalWalls `union` horizontalWalls `difference` holes
```
verticalWalls = fromList \[W.Wall \{W.r = r, W.c = randomCol\} \mid r \leftarrow [(tr + 1)..(br - 1)]\]
horizontalWalls = fromList \[W.Wall \{W.r = randomRow, W.c = c\} \mid c \leftarrow [(tc + 1)..(bc - 1)]\]
topleft = generateWalls tr tc randomRow randomCol g3
topRight = generateWalls tr randomCol randomRow bc g4
bottomLeft = generateWalls randomRow tc br randomCol g5
bottomRight = generateWalls randomRow randomCol br bc g6
(holes, _) = getHoles tr tc br bc randomRow randomCol g7

2.3 Results
Below are the screenshots of the visualization of the generated maze, the program also accepts the third parameter, which is a seed integer provided by users. The idea is when the given seed is the same, the maze generated will be the same as well.
3 Parallelization

For the parallelization, we firstly checked our maze generation algorithms. Because we used recursive division, this algorithm keeps dividing the frames with a horizontal wall and a vertical wall. During this process, we find the possibility of parallelization for the algorithm.

The first step we conducted our experiment is that we run the algorithm in a sequential way to generate a 5000 * 5000 maze and check the result.

```
Registering library for maze-generator-0.1.0.0..
44,440,349,848 bytes allocated in the heap
19,788,555,000 bytes copied during GC
  2,930,768,648 bytes maximum residency (19 sample(s))
  7,742,712 bytes maximum slop
  6426 MiB total memory in use (0 MB lost due to fragmentation)

Gen 0 42674 colls, 0 par 15.930s 16.283s 0.0004s 0.0110s
Gen 1  19 colls, 0 par 12.450s 16.819s 0.8852s 6.9059s

TASKS: 4 (1 bound, 3 peak workers (3 total), using -N1)
SPARKS: 0 (0 converted, 0 overflowed, 0 dud, 0 GC’d, 0 fizzled)

INIT time 0.000s ( 0.004s elapsed)
MUT time 20.831s ( 22.721s elapsed)
GC time  28.300s ( 33.102s elapsed)
EXIT time 0.000s ( 0.007s elapsed)
Total time 49.211s ( 55.834s elapsed)

Alloc rate 2,133,407,274 bytes per MUT second
Productivity 42.3% of total user, 40.7% of total elapsed
```
After finding the total time is about 56s, we tried to use the first method to implement the parallelization. The first method is using par in the Control.Parallel package.
We can find that using par and running the code on 8 cores, that total time decreased to about 37s.

Then, we tried the Strategy and used runEval and rpar to do the parallelization. The result is below.
We can see that using the rpar and running the code on 8 cores, the total time decreased to about 26s, far from the original sequential time of 56s. But, we still have a problem that the GC time is too much and there are a lot of unnecessary sparks generated during the code running. To solve this problem, we tried to control
the depth to parallel to a certain depth. We implemented the depth control like below.

```haskell
generateHWalls 0 tr tc br bc seed = generateWallsOriginal tr tc br bc seed
generateWalls deep tr tc br bc seed
  -- | trace ("generateHWalls for the space: ") ++ show (tx, ty, bx, by)) False = undefined
  -- | trace ("holes: ") ++ show holes) False = undefined
  | bc - tc < 4 || br - tr < 4 = (empty, seed)
  | otherwise = (walls
    `union` runEval (rpar (topLeft `union` topRight))
    `union` runEval (rpar (bottomLeft `union` bottomRight)))
  , newSeed5)

where
  (randomRow, newRowSeed) = pickRandom [(tr + 2), (tr + 4)..(br - 2)] seed
  (randomCol, newColSeed) = pickRandom [(tc + 2), (tc + 4)..(bc - 2)]

newRowSeed
  walls = verticalWalls `union` horizontalWalls `difference` holes
  verticalWalls = fromList [W.Wall {W.r = r, W.c = randomCol} | r <- [(tr + 1)..(br - 1)]]
  horizontalWalls = fromList [W.Wall {W.r = randomRow, W.c = c} | c <- [(tc + 1)..(bc - 1)]]
  (topLeft, newSeed1) = generateWalls (deep - 1) tr tc randomRow randomCol seed
  (topRight, newSeed2) = generateWalls (deep - 1) tr randomCol randomRow bc seed
  (bottomLeft, newSeed3) = generateWalls (deep - 1) randomRow tc br randomCol seed
  (bottomRight, newSeed4) = generateWalls (deep - 1) randomRow randomCol br bc seed
  (holes, newSeed5) = getHoles tr tc br bc randomRow randomCol seed

generateWallsOriginal tr tc br bc seed
  -- | trace ("generateHWalls for the space: ") ++ show (tx, ty, bx, by)) False = undefined
  -- | trace ("holes: ") ++ show holes) False = undefined
  | bc - tc < 4 || br - tr < 4 = (empty, seed)
  | otherwise = (walls
    `union` runEval (rpar (topLeft `union` topRight))
    `union` runEval (rpar (bottomLeft `union` bottomRight)))
  , newSeed5)

where
  (randomRow, newRowSeed) = pickRandom [(tr + 2), (tr + 4)..(br - 2)] seed
```
(randomCol, newColSeed) = pickRandom [(tc + 2), (tc + 4)..(bc - 2)]
newRowSeed
walls = verticalWalls `union` horizontalWalls `difference` holes
verticalWalls = fromList [W.Wall {W.r = r, W.c = randomCol} | r <- [(tr + 1)..(br - 1)]]
horizontalWalls = fromList [W.Wall {W.r = randomRow, W.c = c} | c <- [(tc + 1)..(bc - 1)]]
(topLeft, newSeed1) = generateWallsOriginal tr tc randomRow randomCol seed
(topRight, newSeed2) = generateWallsOriginal tr tc randomRow randomCol randomRow bc seed
(bottomLeft, newSeed3) = generateWallsOriginal randomRow tc br randomCol seed
(bottomRight, newSeed4) = generateWallsOriginal randomRow randomCol br bc seed
(holes, newSeed5) = getHoles tr tc br bc randomRow randomCol seed

After control the depth of the code, we found that the sparks generated decreased a lot.

5 Code Listing

Main.hs

module Main where
import System.Exit(die);
import System.Environment (getArgs, getProgName)
import System.Random (getStdGen)
import System.CPUTime
import Generator
import Control.DeepSeq

main :: IO Integer
main = do
  args <- getArgs
  seed <- getStdGen
  case args of
    [deep, width, height] -> do
      start <- getCPUTime
      let r = mazeGenerator (read deep) (read width) (read height) seed
      end <- r `deepseq` getCPUTime
      return (end - start)
    _ -> do
      progName <- getProgName
      die $ "Usage: " ++ progName ++ " <width> <height>"

Generator.hs (using rpar to parallelize)

module Generator where

import qualified Maze as M
import qualified Wall as W
import Debug.Trace
import Data.Bifunctor
import Control.Monad
import Data.Array.IO
import Data.Set (Set, fromList, union, empty, difference)
import System.Random
import Control.Parallel.Strategies

-- mazeGenerator :: RandomGen g => Int -> Int -> g -> M.Maze
mazeGenerator deep width height seed =
  M.Maze {
    M.m = m,
    M.n = n,
M.walls = initializeFrame m n `union` fst (generateWalls deep 0 0 (m - 1) (n - 1) seed)
    -- M.walls = initializeFrame m n
}
where
    m = width * 2 + 1 -- number of matrix rows
    n = height * 2 + 1 -- number of matrix cols

initializeFrame :: Int -> Int -> Set W.Wall
initializeFrame m n = fromList frame

where
    frame = [ W.Wall {W.r = r, W.c = c} |
       r <- [0..m - 1],
       c <- [0..n - 1],
       r == 0 || r == m - 1 || c == 0 || c == n - 1,
       (r, c) /= (0, 1), -- entrace
       (r, c) /= (m - 1, n - 2)] -- exit

-- Given an frame, generate walls inside the frame
-- top left wall cell: (tr, tc)
-- bottom right wall cell: (br, bc)

generateH Walls 0 tr tc br bc seed = generateWallsOriginial tr tc
    br bc seed

generateWalls deep tr tc br bc seed
    -- | trace ("generateH Walls for the space: " ++ show (tx, ty, bx, by)) False = undefined
    -- | trace ("holes: " ++ show holes) False = undefined
    | bc - tc < 4 || br - tr < 4 = (empty, seed)
    | otherwise = (walls
        `union` runEval (rpar (topLeft `union` topRight))
        `union` runEval (rpar (bottomLeft `union` bottomRight))
        , newSeed5)
    where
        (randomRow, newRowSeed) = pickRandom [(tr + 2), (tr + 4)..(br - 2)] seed
        (randomCol, newColSeed) = pickRandom [(tc + 2), (tc + 4)..(bc - 2)] newRowSeed
        walls = verticalWalls `union` horizontalWalls `difference`
        holes
verticalWalls = fromList [W.Wall {W.r = r, W.c = randomCol} | r <- [(tr + 1)..(br - 1)]]
horizontalWalls = fromList [W.Wall {W.r = randomRow, W.c = c} | c <- [(tc + 1)..(bc - 1)]]
(topLeft, newSeed1) = generateWalls (deep - 1) tr tc randomRow randomCol seed
(topRight, newSeed2) = generateWalls (deep - 1) tr randomCol randomRow bc seed
(bottomLeft, newSeed3) = generateWalls (deep - 1) randomRow tc br randomCol seed
(bottomRight, newSeed4) = generateWalls (deep - 1) randomRow randomCol br bc seed
(holes, newSeed5) = getHoles tr tc br bc randomRow randomCol seed

generateWallsOriginial tr tc br bc seed
-- | trace ("generateHWalls for the space: " ++ show (tx, ty, bx, by)) False = undefined
-- | trace ("holes: " ++ show holes) False = undefined
| bc - tc < 4 || br - tr < 4 = (empty, seed)
| otherwise = (walls `union` runEval (rpar (topLeft `union` topRight))
`union` runEval (rpar (bottomLeft `union` bottomRight)) , newSeed5)
where
  (randomRow, newRowSeed) = pickRandom [(tr + 2), (tr + 4)..(br - 2)] seed
  (randomCol, newColSeed) = pickRandom [(tc + 2), (tc + 4)..(bc - 2)] newRowSeed
  walls = verticalWalls `union` horizontalWalls `difference` holes
  verticalWalls = fromList [W.Wall {W.r = r, W.c = randomCol} | r <- [(tr + 1)..(br - 1)]]
  horizontalWalls = fromList [W.Wall {W.r = randomRow, W.c = c} | c <- [(tc + 1)..(bc - 1)]]
  (topLeft, newSeed1) = generateWallsOriginial tr tc randomRow randomCol seed
  (topRight, newSeed2) = generateWallsOriginial tr randomCol randomRow bc seed
  (bottomLeft, newSeed3) = generateWallsOriginial randomRow tc br randomCol seed
  (bottomRight, newSeed4) = generateWallsOriginial randomRow randomCol br bc seed
generateWallsOriginal randomRow randomCol br bc seed
(holes, newSeed5) = getHoles tr tc br bc randomRow randomCol seed

definitions

getHoles :: RandomGen g => Int -> Int -> Int -> Int -> Int -> Int -> (Set W.Wall, g)
getHoles tr tc br bc rr rc seed =
(fromList $ map (choices !!) $ take 3 $ shuffle seed [0..3],
bottomSeed)

where
randomPick l seed = l !! fst (randomR (0, length l - 1) seed)
choices = [top, left, right, bottom]
(top, topSeed) = pickRandom [W.Wall {W.r = r, W.c = rc| r <-
[(tr + 1), (tr + 3)..(rr - 1)]}] seed
(left, leftSeed) = pickRandom [W.Wall {W.r = rr, W.c = c}| c <-
[(tc + 1), (tc + 3)..(rc - 1)]] topSeed
(right, rightSeed) = pickRandom [W.Wall {W.r = rr, W.c = c}| c <-
[(rc + 1), (rc + 3)..(bc - 1)]] leftSeed
(bottom, bottomSeed) = pickRandom [W.Wall {W.r = r, W.c =
rc}| r <- [(rr + 1), (rr + 3)..(br - 1)]] rightSeed

pickRandom :: RandomGen g => [a] -> g -> (a, g)
pickRandom l seed = Data.Bifunctor.first (l !!) (randomR (0,
length l - 1) seed)

shuffle :: RandomGen g => [a] -> g -> [a]
shuffle gen [] = []
shuffle gen list = randomElem : shuffle newGen newList

where
randomTupple = randomR (0, length list - 1) gen
randomIndex = fst randomTuple
newGen = snd randomTuple
randomElem = list !! randomIndex
newList = take randomIndex list ++ drop (randomIndex+1)
list

Generator.hs (Sequential)
module Generator where

import qualified Maze as M
import qualified Wall as W
import Debug.Trace
import Data.Bifunctor
import Control.Monad
import Data.Array.IO
import Data.Set (Set, fromList, union, empty, difference)
import System.Random

mazeGenerator :: RandomGen g => Int -> Int -> g -> M.Maze
mazeGenerator width height seed =
  M.Maze {
    M.m = m,
    M.n = n,
    M.walls = initializeFrame m n `union` fst (generateWalls 0 0 (m - 1) (n - 1) seed)
    -- M.walls = initializeFrame m n
  }
  where
    m = width * 2 + 1 -- number of matrix rows
    n = height * 2 + 1 -- number of matrix cols

initializeFrame :: Int -> Int -> Set W.Wall
initializeFrame m n = fromList frame
  where
    frame = [ W.Wall { W.r = r, W.c = c} |
      r <- [0..m - 1],
      c <- [0..n - 1],
      r == 0 || r == m - 1 || c == 0 || c == n - 1,
      (r, c) /= (0, 1), -- entrance
      (r, c) /= (m - 1, n - 2)] -- exit

-- Given an frame, generate walls inside the frame
-- top left wall cell: (tr, tc)
-- bottom right wall cell: (br, bc)
generateWalls :: RandomGen g => Int -> Int -> Int -> Int -> g -> (Set W.Wall, g)
generateWalls tr tc br bc seed
  -- | trace ("generateHWalls for the space: " ++ show (tx, ty,
bx, by)) False = undefined
-- | trace ("holes: " ++ show holes) False = undefined
| bc - tc < 4 || br - tr < 4 = (empty, seed)
| otherwise = (walls`union` topLeft
`union` topRight
`union` bottomLeft
`union` bottomRight, newSeed5)
where
  (randomRow, newRowSeed) = pickRandom [(tr + 2), (tr + 4)..(br - 2)] seed
  (randomCol, newColSeed) = pickRandom [(tc + 2), (tc + 4)..(bc - 2)] newRowSeed
  walls = verticalWalls `union` horizontalWalls `difference` holes
  verticalWalls = fromList [W.Wall {W.r = r, W.c = randomCol} | r <- [(tr + 1)..(br - 1)]]
  horizontalWalls = fromList [W.Wall {W.r = randomRow, W.c = c} | c <- [(tc + 1)..(bc - 1)]]
  (topLeft, newSeed1) = generateWalls tr tc randomRow randomCol seed
  (topRight, newSeed2) = generateWalls tr randomCol randomRow bc seed
  (bottomLeft, newSeed3) = generateWalls randomRow tc br randomCol seed
  (bottomRight, newSeed4) = generateWalls randomRow randomRow bc br seed
  (holes, newSeed5) = getHoles tr tc br bc randomRow randomCol seed

getHoles :: RandomGen g => Int -> Int -> Int -> Int -> Int -> Int -> (Set W.Wall, g)
getHoles tr tc br bc rr rc seed =
  (fromList $ map (choices !!) $ take 3 $ shuffle seed [0..3],
  bottomSeed)
where
  randomPick l seed = l !! fst (randomR (0, length l - 1) seed)
  choices = [top, left, right, bottom]
  (top, topSeed) = pickRandom [W.Wall {W.r = r, W.c = rc}| r <- [(tr + 1), (tr + 3)..(rr - 1)]] seed
  (left, leftSeed) = pickRandom [W.Wall {W.r = rr, W.c = c}| c
(- [(tc + 1), (tc + 3)..(rc - 1)]) topSeed
   (right, rightSeed) = pickRandom [W.Wall {W.r = rr, W.c = c}|
   c <- [(rc + 1), (rc + 3)..(bc - 1)]) leftSeed
   (bottom, bottomSeed) = pickRandom [W.Wall {W.r = r, W.c =
   rc}| r <- [(rr + 1), (rr + 3)..(br - 1)]) rightSeed

pickRandom :: RandomGen g => [a] -> g -> (a, g)
pickRandom l seed = Data.Bifunctor.first (l !!) (randomR (0, length l - 1) seed)

shuffle :: RandomGen g => g -> [a] -> [a]
shuffle gen [] = []
shuffle gen list = randomElem : shuffle newGen newList
   where
      randomTuple = randomR (0, length list - 1) gen
      randomIndex = fst randomTuple
      newGen = snd randomTuple
      randomElem = list !! randomIndex
      newList = take randomIndex list ++ drop (randomIndex+1) list

Maze.hs

module Maze where
import qualified Wall as W
import Data.Set (Set)
import Control.DeepSeq

data Maze = Maze {
   m :: Int, -- number of matrix rows
   n :: Int, -- number of matrix cols
   walls :: Set W.Wall
}

instance Show Maze where
   show (Maze m' n' walls') =
      unlines maze
   where
      maze = [rowToString r| r <- [0 .. m'-1]]
      rowToString r = [if isWall r c then '#' else ' ' | c <- [0 ..
n' - 1]]
    isWall r c = W.Wall r c `elem` walls'

instance NFData Maze where
  rnf (Maze m' n' walls') = rnf m' `seq` rnf n' `seq` rnf walls'

Wall.hs

module Wall where
import Control.DeepSeq

data Wall = Wall {
  r :: Int,
  c :: Int
} deriving (Show, Eq, Ord)

instance NFData Wall where
  rnf (Wall r c) = rnf r `seq` rnf c