Parallel Minesweeper Solver

Haoxiang Zhang (hz2763)

December 22, 2021

1 Introduction

Minesweeper is a classic interactive puzzle game that aims to recover all positions of mines without detonating any of them, using clues that tell how many mines are adjacent to a discovered tile. Solving a consistent board has been proven to be co-NP-Complete. [1] Due to the interactive nature of this game, there is a significant portion of a solving process that is going to be sequential, but the hard part of deciding which cells are safe at each step can be sped up using parallel back-tracking. As size of the board grows larger and by deducting multiple safe cells given each board state, the proportion of sequential computations should decrease, giving parallel solver a higher speedup.

2 Problem Formulation

Given a Minesweeper game board with some open tiles, the program should try to find all guaranteed tiles that are safe to click on. If there are no such tiles, the program should return a guess that is one of the least-likely-to-be-a-mine tiles. Repeat this process until all non-mine tiles are opened or when program steps on a mine.

3 Algorithm

3.1 Tank solver algorithm

I used the tank solver algorithm for solving this problem[2]. The algorithm at its core involves enumerating all possible configurations of the mines that satisfies given information on the current board. Then for each tile position, consider among all the possible configurations, how many of them involves the position being a mine. If a tile is not a mine in all possible

[1] Scott, A., Stege, U. & van Rooij, I. Minesweeper May Not Be NP-Complete but Is Hard Nonetheless. Math Intelligencer 33, 5–17 (2011). https://doi.org/10.1007/s00283-011-9256-x

[2] https://luckytoilet.wordpress.com/2012/12/23/2125/

configurations, then the program can safely decide that it is a guaranteed safe tile to click on. But if tiles of such kind doesn't exist, the program will choose the position that is least likely to be a mine, which appears to be a mine in the least number of configurations. The time complexity of this algorithm is $O(2^n)$.

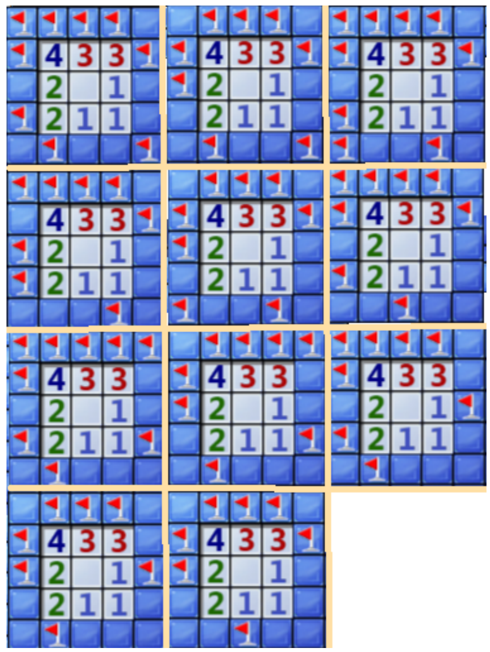

Figure 1: Enumeration of all possible configurations

3.2 Backtracking

To enumerate all possible configurations, the program uses backtracking. First, it needs to find all the connected tiles that is adjacent or diagonally adjacent to at least one open tiles. Each of this connected unopened tiles strip is called a coastal path. For all tiles on the coastal path, we can perform a relative cheap check on whether it being a mine or not violates some of the information from its neighbors. Starting with one end of the path, the program checks if the tile can possibly be a mine given the current information. If yes, it makes the assumption that this tile is a mine and proceeds to the next one; if no, the tile would be assumed not a mine and program proceeds. Upon reaching the end of the path, the program will have found one possible configuration that is all the assumptions used to reach here. It will then alter the last assumption and proceed from there. When both states of a tile has been explored or a conflict is found, the program returns to the previous tile and try to proceed by altering its assumption.

4 Implementation

The input to backtracking function includes state of the game represented as `width`, `height`, `opens` (set of open tile positions as `Point`), `nums` (numbers on open tiles); and progress of current backtracking as unprocessed coastal path and assumed mine positions from processed coastal path. At each step, the function takes the first item out of the unprocessed coastal path, and verifies whether it can be a mine or not. For each possible state of this tile, another backtracking function is called to search for further states, by taking this point out of current unprocessed coastal path and adding it to assumed mine position if applies.

The return value is a tuple of the number of possible configurations for the coastal path, and a map that counts for each point, how many times in all configurations is it not a mine. The ending state is when the unprocessed coastal path is empty, and the function will return an empty map with a 1, indicating that it is a success search of valid configuration. At each step, after the function receives values from the recursive calls, it will merge the maps and sum up values for each tile position if applicable, adding the current point to the map with the number of valid configurations found and return the new tuple.

The parallel implementation is simply creating a spark for each possibility when a tile could potentially be both a mine and not a mine. To prevent too much spark being spawned, this will only take place when the unprocessed length of the coastal path is longer than 5.

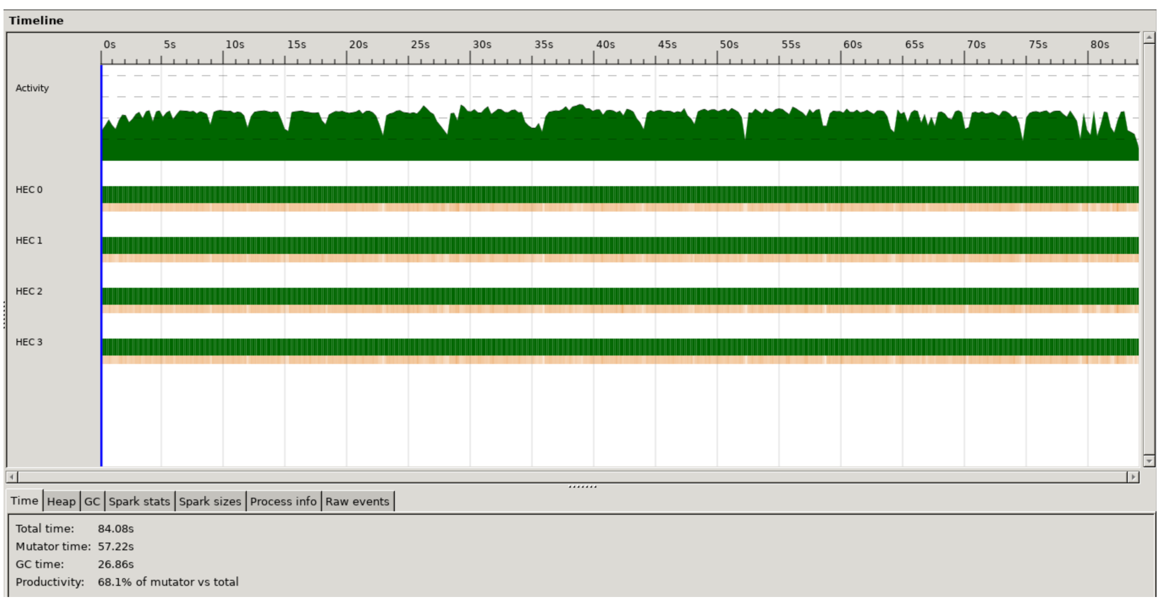

Figure 2: CPU load on Threadscope

5 Performance

The parallel performance is quite dependent on the size of the coastal line and how much of a run is hard to process, which varies a lot from game to game. To benchmark between different number of cores, seeded boards are played across configurations. Table 1 shows the execution time on some seeded boards using different number of cores. The tests are all conducted on a typical "hard" minesweeper board with size 30×16 and 99 mines.

(s)	N1	N2	N4	N8
seed 1	12.89	**10.06**	10.37	15.55
seed 2	19.75	13.64	**10.23**	12.67
seed 3	100.57	95.01	**60.71**	70.73

Table 1: Execution time in parallel

There is some speedup with parallel, but it cannot break even with the overhead of 8 cores, at least not likely on a board of this size. With a easier task, namely a smaller coastal path length at most steps, the speed up of more cores might not be worth the overhead cost. Due to the dynamic creation of sparks, the load balancing between cores is usually fine. Figure 2 shows a sample run with -N4.

6 Code Listing

6.1 Main.hs

```
module Main ( main ) where

import Data.List            ( elemIndex )
import Data.List.Split      ( splitOneOf , chunksOf )
import Data.Char            ( toUpper )

import Data.Set             ( Set )
import Data.Set as Set      ( member , notMember , size , insert , empty , null ,
    foldr , filter)

import Data.Maybe           ( isNothing )
import Control.Monad        ( forM , forM_ , when)

import System.Environment ( getArgs )
import System.Exit          ( exitSuccess)
import System.Random        ( randomR , mkStdGen )

import Util
import MapGenerator         ( minePoints )
import LayoutRender         ( drawPlay , drawOver )
```

```haskell
import AISolver           ( showAllPossibleSafePoints , nextMove )

-- | Calculate the number of surrounding mines for each point.
neighbourMines :: Set Point -> Int -> Int -> [[Int]]
neighbourMines minePs w h = chunksOf w $ map neighbourMinesOf (gridPoints w h)
    where neighbourMinesOf :: Point -> Int
          neighbourMinesOf s = Set.foldr (\p acc -> acc + fromEnum (member p minePs)) 0 (neighboursOf w h s)

-- | Control the whole interactive process during game.
play :: Set Point -> Set Point -> [[Int]] -> Int -> IO ()
play opens minePs nums count = do
    drawPlay opens nums
    let (w, h) = dimension nums
    if Set.null minePs
        then do
            -- Just start at some random point
            let
                gen = mkStdGen 42
                row = fst $ randomR (0, w-1) gen
                col = fst $ randomR (0, h-1) gen
                point   = (3,3)             -- or you could use row col for random start
                minePs' = minePoints w h count point
                nums'   = neighbourMines minePs' w h
            print point
            update opens minePs' nums' count [point,point]
        else do
            let safe_moves = nextMove w h opens nums
            if not $ Prelude.null safe_moves
                then do
                    putStrLn $ showAllPossibleSafePoints w h opens nums
                    update opens minePs nums count safe_moves
                else do
                    putStrLn "No more safe moves.\n"


-- | An extract function from play. This function is responsible for
-- updating the open state of Points and the game layout.
update :: Set Point -> Set Point -> [[Int]] -> Int -> [Point] -> IO ()
update opens minePs nums count points@(point:xs) =
    if point `member` minePs
        then do
            drawOver minePs nums
            putStrLn "Game OVER! You may want to try again?\n"
            exitSuccess
        else do
            let newOpens = reveal point opens minePs nums
                (w, h) = dimension nums
```

```
            if size newOpens == w * h - size minePs
                then do
                    drawPlay newOpens nums
                    putStrLn "Congratulations!\n"
                    exitSuccess
                else do
                    if Prelude.null xs
                        then do play newOpens minePs nums count
                    else do update newOpens minePs nums count xs
update opens minePs nums count points =
    do print "Error in update: no points received"


-- | Handle reveal event, recursively reveal neighbour Points if necessary.
reveal :: Point -> Set Point -> Set Point -> [[Int]] -> Set Point
reveal n opens minePs nums
    | n `member` opens       = opens   -- Point n already opened
    | numAtPoint nums n /= 0 = insert n opens
    | otherwise              = let newOpens = insert n opens
                               in Set.foldr (\p acc -> reveal p acc minePs nums) newOpens
                                     (safeUnopenedNeighbours n newOpens minePs)
        where
            (w, h) = dimension nums

            safeUnopenedNeighbours :: Point -> Set Point -> Set Point -> Set Point
            safeUnopenedNeighbours p opens minePs =
                Set.filter (\nb -> nb `notMember` minePs && nb `notMember` opens) (neighboursOf w h p)


main :: IO ()
main = do
    [width, height, count] <- getArgs
    let w         = read width  :: Int
        h         = read height :: Int
        c         = read count  :: Int

        maxMines  = w * h `quot` 2
        maxHeight = length rows
    if c > maxMines
        then putStrLn $ "Number of mines should less then " ++ show maxMines
        else if h > maxHeight
            then putStrLn $ "Number of rows should no greater then " ++ show maxHeight
            else play empty empty (replicate h (replicate w 0)) c
```

6.2 AISolver.hs

```haskell
module AISolver ( showAllPossibleSafePoints , nextMove ) where

import Util

import Data.Set              ( Set )
import Data.Set as Set       ( foldr , null , notMember , union , intersection , size , toAscList
                             , insert , empty , member , filter )

import Data.Sequence         ( Seq )
import Data.Sequence as Seq ( empty , filter , (<|) , update , index , findIndexL , drop , take
                             , (><) , mapWithIndex , foldrWithIndex , length )
import Data.Foldable         (toList)
import Data.Map              (Map)
import Data.Map as Map       ( empty , singleton , unionWith , union , foldWithKey)
import Control.Monad.Par      ( spawnP , get , runPar)
import Debug.Trace           ( trace)

-- | Find all unrevealed neighbours of an opened Point as a Set for
-- all open Points , and put all the sets in a Sequence.
classifyNeighboursByOpens :: Int -> Int -> Set Point -> Seq (Set Point)
classifyNeighboursByOpens w h opens =
    Seq.filter (not . Set.null) $
        Set.foldr (\p acc -> unrevealNeighboursOf p <| acc) Seq.empty opens
            where unrevealNeighboursOf p = Set.filter (`notMember` opens) (neighboursOf w h p)

-- | Make continuous Points in a group , and return all these groups in a Sequence.
groupContinuousPs :: Seq (Set Point) -> Int -> Seq (Set Point)
groupContinuousPs seq location | location >= Seq.length seq - 1      = seq
                               | Just n <- findGroupNeighbour ed =
                                   groupContinuousPs (Seq.take location seq ><
                                                                           Seq.update n (st `Set.union` (ed `index` n)) ed) location
                               | otherwise                       = groupContinuousPs seq (location + 1)
    where st                   = seq `Seq.index` location
          ed                   = Seq.drop (location + 1) seq
          isContinuous sp1 sp2 = Set.size (sp1 `intersection` sp2) > 0
          findGroupNeighbour   = Seq.findIndexL (isContinuous st)

seqSetToSeqList :: Seq (Set Point) -> Seq [Point]
seqSetToSeqList = Seq.mapWithIndex (\_ sp -> toAscList sp)

-- | Get continuous Points in a group , and return all these groups in a
```

```haskell
    list.
getCoastalPathes :: Int -> Int -> Set Point -> Seq [Point]
getCoastalPathes w h opens = seqSetToSeqList $
    groupContinuousPs (classifyNeighboursByOpens w h opens) 0


getNeighbourOpenNumPs :: Int -> Int -> Point -> Set Point -> [[Int]] -> Set Point
getNeighbourOpenNumPs w h p opens nums =
    Set.filter isOpenNum (neighboursOf w h p)
        where isOpenNum nb = nb `member` opens && numAtPoint nums nb /= 0

-- | The return value is a tuple of the number of all possible mine-location-
--   configurations given the current board and assumptions
--   and a map between points on the path, and how many times among all the
--   configurations it is safe (not a mine)
backtrack :: Int -> Int -> Set Point -> [[Int]] -> [Point] -> Set Point -> (Map Point Int, Int)
backtrack w h opens nums [] mineFlags   = (Map.empty, 1)
backtrack w h opens nums (x:xs) mineFlags = do
    let
        couldbeMine = verify x True
        couldbeFine = verify x False
    if couldbeMine && couldbeFine && Prelude.length xs > 5 then
        runPar $ do
            spark_m <- spawnP $ backtrack w h opens nums xs (x `insert` mineFlags)
            spark_f <- spawnP $ backtrack w h opens nums xs mineFlags
            (mine_map, possible_count_m) <- get spark_m
            (fine_map, possible_count_f) <- get spark_f
            let fine_map' = Map.union fine_map $ Map.singleton x possible_count_f
            return (unionWith (+) mine_map fine_map', possible_count_m + possible_count_f)
    else do
        let
            (mine_map, possible_count_m)    | couldbeMine = backtrack w h opens nums xs (x `insert` mineFlags)
                                            | otherwise = (Map.empty, 0)
            (fine_map, possible_count_f)    | couldbeFine = backtrack w h opens nums xs mineFlags
                                            | otherwise = (Map.empty, 0)
            fine_map' = Map.union fine_map $ Map.singleton x possible_count_f

        (unionWith (+) mine_map fine_map', possible_count_m + possible_count_f)
    where nbOpenNumPs p             = getNeighbourOpenNumPs w h p opens nums
```

```haskell
          numMineFlagInNeighbours p = Set.size $ Set.filter (`member` mineFlags) (neighboursOf w h p)
          numUnknowNeighbours p     =
              Prelude.length $ Prelude.filter (`member` neighhours) xs
                  where neighhours = neighboursOf w h p
          verify :: Point -> Bool -> Bool
          verify p@(r, c) isMine = Set.foldr (\p acc -> acc && verifyNum p isMine) True (nbOpenNumPs p)

          verifyNum :: Point -> Bool -> Bool
          verifyNum p isMine = numMineFlagInNeighbours p + fromEnum isMine <= numAtPoint nums p &&
                                  numAtPoint nums p <= numMineFlagInNeighbours p +
                                                         fromEnum isMine +
                                                         numUnknowNeighbours p


nextMove :: Int -> Int -> Set Point -> [[Int]] -> [Point]
nextMove w h opens nums = do
    let coastalPaths        = getCoastalPathes w h opens
        backtrackResults    = map (\path -> backtrack w h opens nums path Set.empty) $ toList coastalPaths
        allSafePoints       = Prelude.foldr (\(p_c, p_t) l -> l ++ getSafePoints p_c p_t) [] backtrackResults
    if Prelude.null allSafePoints
        then let res@(point, acc) = Prelude.foldr (\(p_c, p_t) candi -> getBestGuess p_c p_t candi) ((0,0),0/1 :: Float) backtrackResults in [point]
        else allSafePoints
    where
        getSafePoints :: Map Point Int -> Int -> [Point]
        getSafePoints p_counts p_total = foldWithKey (\k v l -> if v == p_total then k:l else l) [] p_counts
        getBestGuess :: Map Point Int -> Int -> (Point, Float) -> (Point, Float)
        getBestGuess p_counts p_total candidate = foldWithKey (\k v (cp, cv) -> if (fraction v p_total > cv) then (k, fraction v p_total) else (cp, cv)) candidate p_counts
            where fraction a b = (fromIntegral a) / (fromIntegral b)

-- | Change to all possible safe Points to printable string.
showAllPossibleSafePoints :: Int -> Int -> Set Point -> [[Int]] -> String
showAllPossibleSafePoints w h opens nums =
    "All safe locations: " ++ (show $ map pointToLoc points) ++ "\n"
        where points            = nextMove w h opens nums
              pointToLoc (r, c) = (rows !! r, c)
```

6.3 MapGenerator.hs

```haskell
module MapGenerator ( minePoints ) where

import Util

import Data.Set         ( Set )
import Data.Set as Set  ( size, insert, empty, member )

import System.Random    ( randomRs, newStdGen, mkStdGen, setStdGen, getStdGen, split )

import System.IO.Unsafe ( unsafePerformIO )

-- | Generate all the mine Points excluding the initial point.
minePoints :: Int -> Int -> Int -> Point -> Set Point
minePoints w h count point = collect empty (rands w h)
    where  -- Collect mines non-repetitive Points.
            collect :: Set Point -> [Point] -> Set Point
            collect ps (x:xs)
               | size ps >= min count (w * h - 1) = ps  -- the max available mine positions
                                                         -- is w * h - 1, where 1 is the
                                                         -- initial point
               | otherwise                        = if x `member` ps || x == point
                                                     then collect ps xs
                                                     else collect (insert x ps) xs

-- | Produce an infinite list of random Points
rands :: Int -> Int -> [Point]
rands w h =
    let
        -- for benchmarking use seeded mkStdGen
        (gw, gh) = split $ unsafePerformIO $ newStdGen -- mkStdGen 10342
        rs       = randomRs (0, w - 1) gw
        cs       = randomRs (0, h - 1) gh
    in zip cs rs
```

6.4 LayoutRender.hs

```haskell
module LayoutRender ( drawPlay, drawOver ) where

import Util

import Text.Tabular
import Text.Tabular.AsciiArt ( render )
```

```
import Data.List.Split       ( chunksOf )

import Data.Set              ( Set )
import Data.Set as Set       ( member )

-- | Put constructor Header on each element in sourceList.
rangeHeader :: Int -> [String] -> [Header String]
rangeHeader len sourceList = take len $ map Header sourceList

-- | Draw the game's layout according to the convert function and
-- the array of number of neighbour mines of each Point.
draw :: (Point -> String) -> [[Int]] -> IO ()
draw convert nums = putStr $ render id id id gridLayout
    where (w, h)    = dimension nums

          grid :: [[Point]]
          grid = chunksOf w $ gridPoints w h

          gridLayout :: Table String String String
          gridLayout = Table
              (Group SingleLine
                  [ Group SingleLine $ rangeHeader h [[c] | c <- rows] ])
              (Group SingleLine
                  [ Group SingleLine $ rangeHeader w [show n | n <- [0..]] ])
              (map (map convert) grid)

-- | Draw the game's layout according to the open Points and
-- the array of number of neighbour mines of each Point.
drawPlay opens nums = draw convert nums
    where -- Convert a Point position to its representation,
          -- either black block or number of neighbour mines.
          convert :: Point -> String
          convert p | p `member` opens = show $ numAtPoint nums p
                    | otherwise        = ['\x2588']

-- | Draw the game over layout.
drawOver :: Set Point -> [[Int]] -> IO ()
drawOver minePs nums = draw convert nums
        where -- Convert a Point position to its representation in
              -- String, either black block or number of neighbour mines.
              convert :: Point -> String
              convert p | p `member` minePs = "*"
                        | otherwise         = show $ numAtPoint nums p
```

6.5 Util.hs

```
module Util where

```

```haskell
import Data.Set        ( Set )
import Data.Set as Set ( fromList, member, filter )

type Point = (Int, Int)

-- | Generate a list of Point (r, c) according to the width and height
-- of the layout in the game, with r is the index of row
-- and c is the index of column.
gridPoints :: Int -> Int -> [Point]
gridPoints w h = [(r, c) | r <- [0 .. h - 1], c <- [0 .. w - 1]]

-- | Get tuple (width, height) from a two dimension list.
dimension :: [[Int]] -> (Int, Int)
dimension a = (length $ head a, length a)

rows = ['A'..'Z']

-- | Get neighbours of a Point in a Set.
neighboursOf :: Int -> Int -> Point -> Set Point
neighboursOf w h (r, c) = Set.filter (`member` gridPs) possibleNeighbours
    where gridPs             = fromList $ gridPoints w h
          possibleNeighbours = fromList [(r, c - 1), (r, c + 1),
                                         (r + 1, c), (r + 1, c + 1), (r + 1, c - 1),
                                         (r - 1, c), (r - 1, c + 1), (r - 1, c - 1)]

numAtPoint nums (r, c) = nums !! r !! c
```