Mohsin Rizvi
COMS 4995 - Parallel Functional Programming
Fall 2024
Final Report

Parallel Maze Solver Using A*

Mohsin Rizvi

Introduction: Mazes and A*

For my project, I implemented a maze-solving program built on top of the A* path-finding algorithm. The program's command-line usage is as follows:

```
mazeSolver <mazefile> <depth> [-show] +RTS -N<depth>
```

The program takes in a path to a *maze file*, which is a text file containing a two-dimensional maze. Each character in the maze represents a *maze node* or *maze tile*, which is simply a location in a maze. The valid characters in the maze file (in additional to newline characters) are as follows:

- An '`s`' denotes the start node of a maze, where the maze solution must start from.
- A '`g`' denotes the goal node of a maze, where the maze solution must end up.
- A ' ' (a space) denotes an empty node in the maze, which a maze solution path may go through.
- An '`X`' denotes a wall node in the maze, which a maze solution path may NOT go through.

All rows in the maze must have the same length. The following is a valid example maze:

```
X  X
 X  XX
  X    X X
        XX
gXX  X X
XXX
XXXX XX  X
XXXXXX  XX
XXXXXXX Xs
XXXXXX
```

The program also takes in a user-specified *depth*, or level of parallelism. By default, the program prints out the coordinates of the resulting path through the maze. The user may also specify the `-show` argument, in which case the program will print back the maze with the path overlaid on top of it. The path is denoted using '.' characters connecting the start and goal nodes.

The program attempts to find a close-to-optimal path through the maze from the start node to the goal node. The program refers to maze nodes by their coordinates, with the top-left node of the maze having coordinates (0, 0). The path may not go diagonally from one node to another; it may only go up, down, left, or right. The maze must have exactly one start tile, exactly one goal tile, and all empty tiles must be reachable from all other empty tiles (i.e., there are no "islands" of empty tiles in the maze separated entirely by wall tiles). The output from running the program on the earlier example maze with the -show argument is as follows:

```
10x10
start: (9,8)
goal:  (0,4)

Path (length 18):
[(9,8),(9,9),(8,9),(7,9),(7,8),(7,7),(7,6),(7,5),(6,5),(5,5),(4,5),(3
,5),(3,4),(3,3),(2,3),(1,3),(0,3),(0,4)]

X  X
 X  XX
  X    X X
....    XX
gXX. X X
XXX.....
XXXX XX. X
XXXXXX .XX
XXXXXXX.Xs
XXXXXX ...
```

Note that I changed the characters used to represent maze tiles since my original project proposal, since I found that the original characters were hard to easily differentiate from each other at-a-glance when viewing the maze and verifying a solution.

A* Algorithm

A* is an algorithm that seeks to find the shortest path between two nodes in a graph. It utilizes a heuristic function to estimate the distance from any given node to the goal node. The pseudocode for the algorithm is roughly as follows, given a start node *s*, a goal node *g*, and a heuristic function *h*:

```
- Create a priority queue pq containing element s with priority h(s). This is also called the open set
- Create a mapping pmap from nodes to their previous nodes in the shortest path found
- Create a mapping gmap from nodes to their shortest path distance from the start node, with s
  initially mapping to 0
- While pq is not empty:
    - Extract lowest-priority node cur from pq
    - If cur == g:
        - Recursively use pmap to reconstruct the path from g to s, reverse it, and return
          the reversed version
    - For each neighboring node n of g:
        - If n is not a key in gmap or gmap[cur] + 1 is less than gmap[n]:
            - Set pmap[n] = cur
            - Set gmap[n] = gmap[cur] + 1
            - If n is not in pq, then insert it with priority gmap[n] + h(n)
    - If pq is empty, there is no valid path; return the empty path
```

Since tiles in the maze are represented using coordinates, I used the Euclidean coordinate distance between a given node and a goal node as the heuristic value. This is given by the equation $distance = \sqrt{(x1 - x2)^2 + (y1 - y2)^2}$ for two nodes with coordinates (x1, y1) and (x2, y2).

Implementation and Parallelization Strategies

I started off by implementing the serial version of my maze solver. This largely consisted of parsing the maze file to an internal representation that I could easily use during the actual traversal, including allowing access to maze nodes by their coordinates, and then implementing the actual A* algorithm that operates on the maze.

Since A* requires a priority queue to sort nodes by their heuristic value, I used the PSQueue Haskell package, which implements a priority search queue. I chose to use this package over other priority queue packages, since it implements a priority search queue with O(log n) lookup time. This made it faster to check whether maze nodes had already been added to the priority search queue. Additionally, I chose to use a vector of vectors to represent the two-dimensional grid of maze tiles. I chose this over a more graph-like representation since the grid of tiles is dense, and it was easier for me to reason about and construct the internal maze representation. I also chose to use vectors over lists since my algorithm would rely heavily on accessing nodes by their coordinates in the maze, and vectors allow O(1) access to an indexed element.

After I had a working serial implementation that passed my test cases, I set out to parallelize it. A* for finding optimal paths is difficult to parallelize, since it is impossible to know if a path is optimal until you've traversed the entire length of the path from the start to the goal and checked the heuristic values of all neighboring tiles along the path. In light of this, I spent a bit of time thinking about different strategies for parallelization and deciding whether they would likely lead to a significant speedup. There were two main strategies that I felt were most likely to speedup the program the most, which I will refer to as "multiple starts" and "checkpoint partitioning". For each of these strategies, the program takes in a user-specified level of parallelism.

I also considered a third parallelization strategy, where I would do a modified A* search where multiple threads would process maze tiles from the priority queue in parallel. However, I decided against this as it required shared access to data structures by different threads, and I was unsure whether it would lead to any speedup.

Mohsin Rizvi
COMS 4995 - Parallel Functional Programming
Fall 2024
Final Report

Parallel Strategy: Multiple Starts

This parallel strategy was inspired by Zaghloul et al [1]. In this strategy, the program would first identify a series of candidate "pseudo-start" maze nodes near the "true start" node. The amount of pseudo-start nodes was equal to the user-specified parallelism level. After finding the pseudo-start nodes, I would run an A* search from each of them, each running on its own thread. I implemented this by using `parList` with `rseq` to evaluate each search in parallel. The resulting path from each search was then stitched together with the path from the true start to the pseudo-start, and the shortest path to the goal was taken.

While this solution was simple enough to implement, it did not lead to much of a noticeable speedup. I believe there are two reasons for this:

1) The pseudo-start nodes were generally clustered near each other. Because of this, the searches were often "bottle-necked" through the same points in the maze, and ended up doing a lot of identical work.
2) Each thread still needed to do an (almost) full A* search to the goal node, which meant that the user still has to wait about the same time as it would take a serial A* to finish.

After this strategy failed to yield any impressive results, I went back to the drawing board.

Parallel Strategy: Checkpoint Partitioning

This is a parallel strategy that I came up with after considering why the multiple starts strategy had failed. In this strategy, I attempt to subdivide the search into a number of subsearches, with the amount of subsearches being the user-specified parallelism level. My thinking was that by having each thread do a smaller search, and then combining the results, the program would be able to finish much quicker.

With this strategy, given a user-specified `depth`, we first come up with `depth - 1` "checkpoints." These are simply points that are as close to equally spread-out as possible between the start and goal nodes. These are computed by using the coordinates of the start and goal nodes. After we compute candidate checkpoint locations, we check the tile type of the candidate checkpoint. If the checkpoint is an empty tile (i.e., a valid part of a path), then we keep it. If it is a wall tile (i.e., not a valid part of a path), then we do a breadth-first search from that tile until we find the closest empty tile, and we use that instead.

After we have our list of checkpoints, we prepend the list with the start node, and append it with the goal node. Next, we create `depth` overlapping pairs of nodes from this list to represent `depth` subsections of a theoretical path. At this point, we run A* in parallel on each pair, with the first member of the pair being a start node and the second member being a goal node. These A* searches are evaluated in parallel by using `parList` with `rseq` over a list of the node pairs, with each parallel thread handling one pair.

After this point, if the goal node is reachable, we can glue the resulting paths together end-to-end to create a valid path from the start node to the goal node. After my initial implementation of this strategy, however, I noticed a few things about the resulting paths:

1) There were adjacent duplicate nodes in the final path listing, since one subsection's goal node would be the same as the next subsection's start node.
2) There were points where checkpoints were slightly off the optimal path, and the resulting path would go out of its way to reach a checkpoint, before doubling-back to its original path.
3) If the goal was unreachable, a partial path was incorrectly returned as a solution, since some of the subsearches still completed successfully.

To address these, I added a few post-processing steps on the resulting path. These steps remove adjacent duplicates, remove unnecessary offshoot branches from the path (by checking for non-adjacent duplicate tiles in the path), and verify that the solution is a valid path (i.e., all path members are adjacent to their neighboring path members, and the path ends with the ultimate goal node). After I implemented these extra steps, I found that the resulting path from the start to the goal was correct, and significantly closer in length to the optimal path.

The Result

I ended up going with checkpoint partitioning as my parallelization strategy, since it led to a *far* bigger speedup in my test cases, and the percent increase in path length was minimal relative to the increase in speed after implementing some optimizations and post-processing steps. In the end, I ended up with a program that successfully computes a path from the start tile to the goal tile in parallel, and it generally does it very fast when given more cores (see "Performance" below for more information on speed). To reiterate, the program takes a user-specified level of parallelism via a command-line argument, and the user can also specify an optional `-show` flag to print out the resulting path rendered on the maze (as opposed to the default, which just shows a list of the coordinates that make up the path).

Note that due to the nature of the checkpoint partitioning strategy, the path found when using parallelism is not guaranteed to be optimal (i.e., the shortest possible path), since some of the checkpoints may be slightly out of the way. However, I felt that this was a reasonable tradeoff to make, since parallelization made the program much faster, the resulting paths were not terribly longer than optimal, and I was able to take some steps to mitigate the increase in path length. Additionally, if there are disjoint (i.e., nonreachable) parts of the maze, then it is possible for the checkpoint parallelism strategy to incorrectly say that no path exists if one of the checkpoints is not reachable from the start and/or goal nodes. Because of this, the user-supplied maze should not have disjoint sections.

Testing

Mohsin Rizvi
COMS 4995 - Parallel Functional Programming
Fall 2024
Final Report

Initially, I started by drawing a few maze files by hand to handle a few different edge cases as I worked on the serial version of my maze solver. After starting the parallel implementation, however, I realized that I would need very large mazes to more clearly measure the performance difference from using more cores. To do this, I wrote a small Python3 program (called `mazegen.py`) to generate mazes with user-specified dimensions. Using this, I was able to quickly generate large mazes with hundreds of thousands of tiles that I could use to test performance.

A selection of mazes generated by `mazegen.py` that I used for testing are included in my submission. Some of these mazes are hand-modified versions of generated mazes that I edited to test out different edge cases. The `mazegen.py` program is also included in my submission and is also pasted in this report further down.

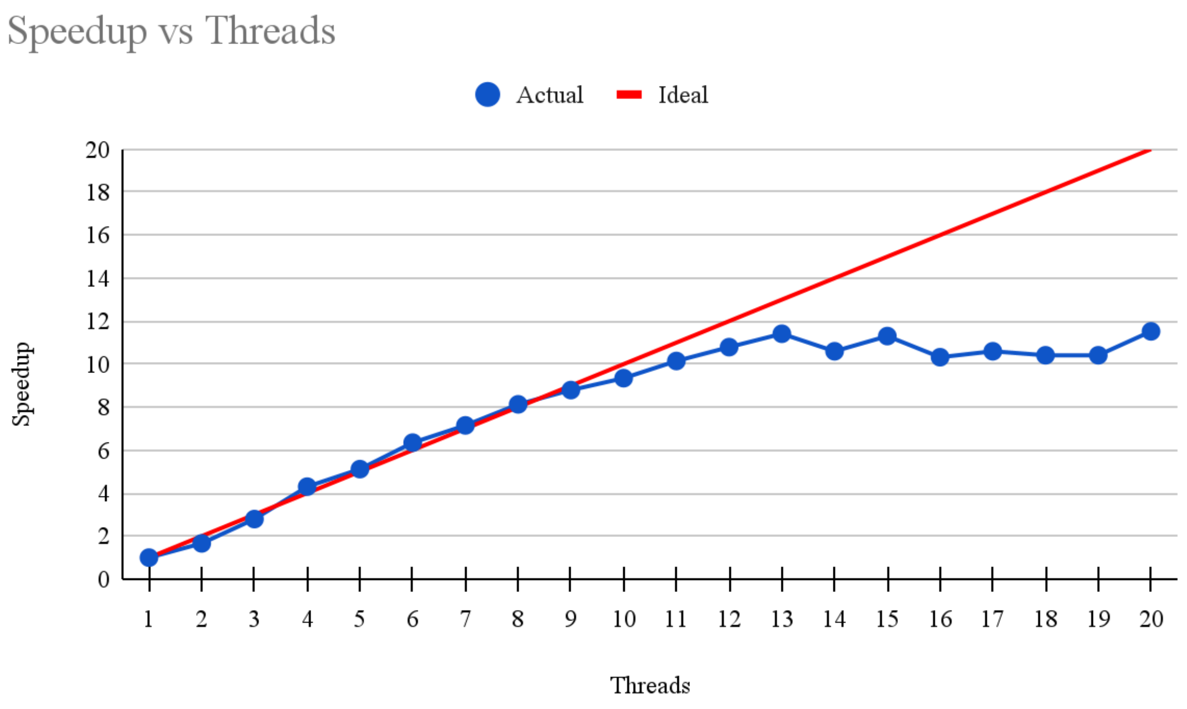
Performance

The following performance measurements were made on my Windows PC with an Intel Core i5-14600K processor with 20 cores. Note that all performances noted below are measured by running the program without the `-show` option, so that IO costs to print the maze (which may be large) aren't included in the measured program runtime. Additionally, I used the maze file 200x1000.txt, which is included in my submission under the test/ directory. This is a maze with dimensions 200 by 1000 tiles (for a grand total of 200,000 tiles) that I generated using `mazegen.py`.

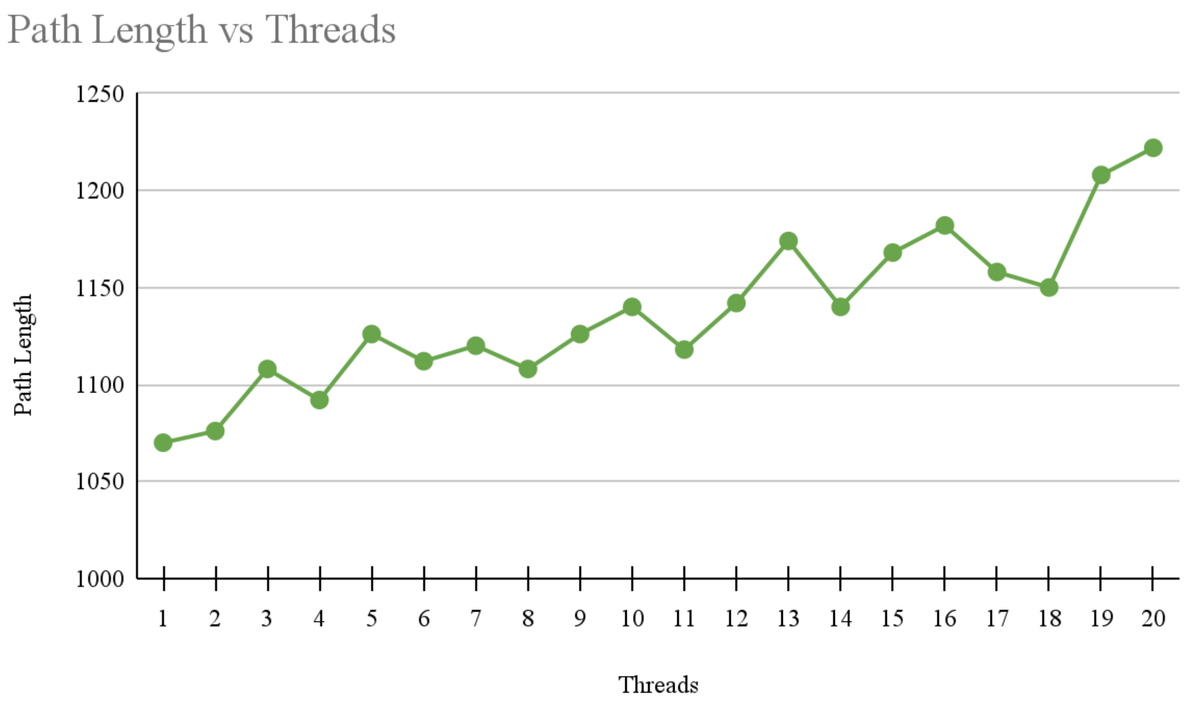
The following table and two line charts show the performance of the parallelized program on 1 through 20 cores, including the speedup and path length results when running the maze solver with a given degree of parallelism.

Degree of Parallelism	Runtime (seconds)	Path Length
Serial: 1	1.187	1070
2	0.714	1076
3	0.425	1108
4	0.276	1092
5	0.232	1126
6	0.187	1112
7	0.166	1120
8	0.146	1108
9	0.135	1126

10	0.127	1140
11	0.117	1118
12	0.110	1142
13	0.104	1174
14	0.112	1140
15	0.105	1168
16	0.115	1182
17	0.112	1158
18	0.114	1150
19	0.114	1208
20	0.103	1222



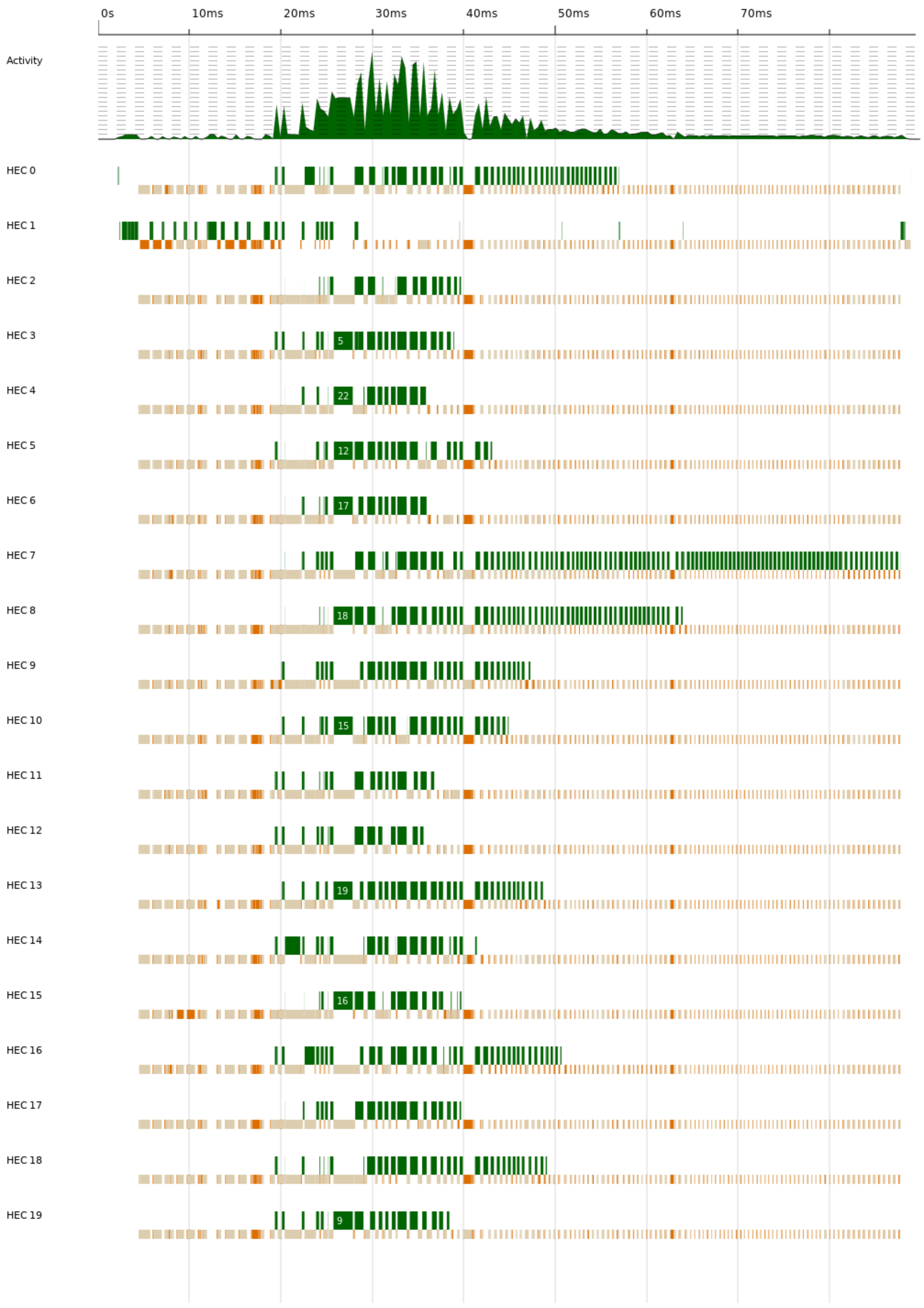
Mohsin Rizvi
COMS 4995 - Parallel Functional Programming
Fall 2024
Final Report



As you can see, the speedup from adding more cores seems to scale linearly until around ~12 cores, at which point the benefits from adding more parallelism begin to decrease. The path length seems to continue climbing with more parallelism, though I suspect larger mazes with more space between the start and goal tiles would lead to less of an increase in path length added per core. That being said, the speedup from using 12 cores is relatively large compared to the increase in path length from 12 cores (a ~10.7x speedup compared to a 6% increase in path length), so I think this is a reasonable tradeoff. In a practical application, I would recommend against running the parallel solution with more than 12 cores against mazes similar to the one used here for performance measuring.

The below Threadscope screenshot shows core activity across the program runtime when run with 20 cores. As you can see, the workload seems to be fairly well-distributed around the second quarter of the program's runtime, but as time goes on most of the threads are waiting on one or two other long-running threads to finish. Due to the fact that we are parallelizing only by subdividing an ideal path through the maze, it is likely that the workload distributed to the threads are not actually even. As a result, I suspect the workload balance would vary a lot by maze when using this parallelization strategy.

Mohsin Rizvi
COMS 4995 - Parallel Functional Programming
Fall 2024
Final Report



Mohsin Rizvi
COMS 4995 - Parallel Functional Programming
Fall 2024
Final Report

Code

The code for the parallel maze solver is as follows, including comments on usage and compilation instructions. This file is also submitted as `mazeSolver.hs`.

```
{-
compile with:
$ stack install vector
$ stack install PSQueue
$ stack install monad-par
$ stack --resolver lts-22.33 ghc -- --make -Wall -O mazeSolver.hs -threaded
-rtsopts

run with:
./mazeSolver <mazefile> <depth> [-show] +RTS -N<depth>

My project is a parallel implementation of A* that uses the algorithm to solve
mazes.
As input, it takes the path to a "maze file" which contains an ASCII
representation of a maze.
A maze is a 2-dimensional grid of tiles, where each tile can be one of the
following:
- ”s‟ denotes a start tile that may be part of a maze solution
- “g” denotes a goal tile that may be part of a maze solution
- “ ‟ denotes an empty tile that may be part of a maze solution
- “X” denotes a wall tile that may not be part of a maze solution

A maze must contain exactly one start tile and one goal tile. Paths cannot go
diagonally between
tiles, and all empty tiles must be reachable from all other empty tiles (i.e.,
there are no
"islands"). For example, the following 4x4 grid is a valid maze:
s  X
 X X
X
g X

My project reads in the maze file, constructs an in-memory representation of a
maze, and runs the
parallelized A* algorithm on the maze. The output from the program is a path
from the
start tile to the goal tile (with the path represented as an array of
coordinates), or a message
indicating that no solution path exists. You can use the "-show" flag to have
the program print
out the maze with the path annotated on it. For my A* heuristic function, I use
the distance
between a given tile and the goal tile, computed using the X and Y positions of
tiles in the maze.

Parallelism strategies attempted:
"Checkpoint partitioning": This is a strategy I came up with that I decided to
use in my final
version of the project. This involves finding <depth - 1> "checkpoints" in a
line between the
start and goal tiles, then running A* on each closest pair of points to get
partial paths, before
finally stitching the paths together and smoothing out any redundant "bumps" in
the path. This is
the fastest approach I tried, but can lead to a slightly less optimal path. I
found that the percent
increase in path length was relatively small compared to the percentage speedup
when running
in parallel.

"Multiple starts": Expand outward from the start node until you find <depth>
nodes from
which to do a search to the goal node. Take the resulting shortest path. I
found that this was not
much faster than a serial A* on most mazes, since the nodes I searched from
were not far apart from
each other and ended up doing almost the same work as a serial A* per thread.

For references used, see the "References" section in the report.
-}

import System.Environment
import System.Exit
import Data.Vector as Vec (Vector, foldr, fromList, (!))
import Data.PSQueue as PSQueue
import qualified Data.Map as Dict
import Data.Maybe
import Data.Set as Set (fromList, member)
import Control.Parallel.Strategies

data TileType = Step | Start | Goal | Wall
instance Show TileType where
  show Step  = " "
  show Start = "s"
  show Goal  = "g"
  show Wall  = "X"
instance Eq TileType where
```

```
  Step == Step   = True
  Start == Start = True
  Goal == Goal   = True
  Wall == Wall   = True
  _ == _         = False

data Node = Node
    { ttype :: TileType
    , x     :: Int
    , y     :: Int
    }
instance Eq Node where
  n1 == n2 = (((x n1) == (x n2)) && ((y n1) == (y n2)))
instance Ord Node where
  compare n1 n2 =
    compare ((show (x n1)) ++ "x" ++ (show (y n1))) ((show (x n2)) ++ "x" ++
(show (y n2)))
instance Show Node where
  show n = show (ttype n)

data Maze = Maze
    { rows  :: Vector (Vector Node)
    , xlen  :: Int
    , ylen  :: Int
    , start :: (Int, Int)
    , goal  :: (Int, Int)
    }

-- uncomment lines from this function to print out the actual maze (without a
path).
instance Show Maze where
  show m = do
    let line1 = (show (xlen m)) ++ "x" ++ (show (ylen m)) ++ "\n"
    let line2 = "start: " ++ show (start m) ++ "\ngoal:  " ++ show (goal m)
    --let linen = Vec.foldr (\a b -> "\n" ++ (Vec.foldr (\c d -> (show c) ++ d)
[] a) ++ b) "" (rows m)
    line1 ++ line2 -- ++ linen

showPath :: Maze -> [(Int, Int)] -> Bool -> [Char]
showPath _ [] _ = "No path exists"
showPath m path show_maze =
  "Path (length " ++ (show (length path)) ++ "): " ++ (show path) ++ "\n" ++
    (case show_maze of
       True -> solution
       _ -> "")
  where
```

```
    solution = Vec.foldr (\a b -> "\n" ++ (Vec.foldr (\c d -> (showWithPath c)
++ d) [] a) ++ b) "" (rows m)
    pathset = Set.fromList path
    showWithPath node =
      case (ttype node) of
        Step -> case (Set.member (x node, y node) pathset) of
                  True -> "."
                  _ -> show node
        _ -> show node

-- heuristic for A*: coordinate distance to the goal node
h :: Maze -> (Int, Int) -> Double
h m (x', y') = do
  let xdiff = (x' - (fst (goal m)))
  let ydiff = (y' - (snd (goal m)))
  let x2 = xdiff * xdiff
  let y2 = ydiff * ydiff
  sqrt ((fromIntegral x2) + (fromIntegral y2))

-- deletes adjacent duplicates from a list
dedup :: Eq a => [a] -> [a]
dedup (x:y:zs) =
  if (x == y)
    then dedup (y:zs)
    else x:(dedup (y:zs))
dedup xs = xs

-- given a list, deletes all elements between nonadjacent duplicates in the
list.
-- for example, smooth [1, 2, 3, 4, 5, 2, 3] == [1, 2, 3].
-- this is used to improve paths returned by the parallel A*.
smooth :: Ord a => [a] -> [a]
smooth as = smoothHelper Dict.empty (0 :: Int) as []
  where
    smoothHelper _ _ [] q = map (\(a, _) -> a) (reverse q)
    smoothHelper dict i (x:xs) q =
      if (Dict.member x dict)
        then do
          let from = fromJust (Dict.lookup x dict)
          smoothHelper dict (i + 1) xs (filter (\(_, b) -> b <= from) q)
        else smoothHelper (Dict.insert x i dict) (i + 1) xs ((x, i):q)

-- generate <depth> pairs of start and goal nodes to use for parallelizing the
algorithm.
checkPoints :: Maze -> Int -> Node -> Node -> [(Node, Node)]
checkPoints m depth start goal = do
  let x1 = x start
  let x2 = x goal
  let y1 = y start
  let y2 = y goal
  let xdiff = abs (x1 - x2)
  let ydiff = abs (y1 - y2)
  let xdiv = div xdiff depth
  let ydiv = div ydiff depth

  let xop = if (x1 < x2) then (+) else (-)
  let yop = if (y1 < y2) then (+) else (-)
  let candidatePoints = [n | i <- [1..(depth - 1)], let x = x1 `xop` (xdiv *
i), let y = y1 `yop` (ydiv * i), let n = findNodeByIndex m (x, y)]
  let middle = dedup (take (depth - 1) (map (\a -> nearestPoint [a])
candidatePoints))
  zip (start:middle) (middle ++ [goal])
  where
    nearestPoint (n:ns) =
      if ((ttype n) == Wall)
        then nearestPoint (ns ++ (allNeighborsOf m n))
        else n
    nearestPoint [] = goal

-- verifies that a path is legal (i.e., that all adjacent steps are one step
apart),
-- and that it ends with the goal
verifyPath :: [(Int, Int)] -> Node -> Bool
verifyPath [] _ = False
verifyPath xs goal = (not (Prelude.null xs)) && (snd (Prelude.foldr
areNeighbors ((-1, -1), True) xs)) && ((last xs) == (x goal, y goal))
  where
    areNeighbors a (b1, b2) = (a, b2 && (isNeighborOf a b1))
    isNeighborOf (a1, a2) (b1, b2) =
      if ((a1, a2) == (-1, -1))
        then True
        else if ((b1, b2) == (-1, -1))
          then True
          else elem (a1, a2) [(b1, b2 - 1), (b1, b2 + 1), (b1 - 1, b2), (b1 +
1, b2)]

-- launch a* in parallel to find a path from the start to goal
aStarPar :: Maze -> Int -> [(Int, Int)]
aStarPar m depth =
  if (depth <= 1)
    then aStar m startN goalN
    else do
      let path = smooth (concat ((parHelper (checkPoints m depth startN goalN))
`using` parList rseq))
```

```
      if (verifyPath path goalN)
        then path
        else []
  where
    startN = findNodeByIndex m (start m)
    goalN = findNodeByIndex m (goal m)
    parHelper ns = [x | (sn, gn) <- ns, let x = aStar m sn gn]

-- given a maze, a start node, and a goal node, use A* to find the shortest
path
aStar :: Maze -> Node -> Node -> [(Int, Int)]
aStar m startN goalN = do
  let oset = PSQueue.singleton startN (h m (x startN, y startN))
  let prev = Dict.empty -- maps Nodes to previous Nodes in best path
  let gmap = Dict.singleton startN (0 :: Int) -- maps Nodes to Ints
  step oset prev gmap
  where
    step openset prev gmap = do
      let (nbinding, openset') = fromJust (minView openset)
      let n = key nbinding
      if (n == goalN)
        then reverse ((x n, y n):(rebuildPath prev n))
        else do
          let neighbors = neighborsOf m n
          let (openset'', prev', gmap') = updateStructures openset' prev gmap n
neighbors
          if (PSQueue.null openset'')
            then []
            else step openset'' prev' gmap'

    rebuildPath prev n = if (n == startN) -- returns [(Int, Int)]
      then []
      else do
        let laststep = fromJust (Dict.lookup n prev)
        (x laststep, y laststep):(rebuildPath prev laststep)

    updateStructures openset prev gmap _ [] =
      (openset, prev, gmap)
    updateStructures openset prev gmap cur (n:ns) = do
      let tmpscore = fromJust (Dict.lookup cur gmap) + 1 :: Int
      if (tmpscore < (Dict.findWithDefault 999999 n gmap)) -- reasonable max
heuristic value
        then do
          let prev' = Dict.insert n cur prev
          let gmap' = Dict.insert n tmpscore gmap
          let openset' = case (PSQueue.lookup n openset) of
```

```
               Nothing -> PSQueue.insert n ((fromIntegral tmpscore) + (h m (x
n, y n))) openset
               _ -> openset
          updateStructures openset' prev' gmap' cur ns
        else
          updateStructures openset prev gmap cur ns

findNodeByIndex :: Maze -> (Int, Int) -> Node
findNodeByIndex m (x, y) = ((rows m)!y)!x

neighborsOf :: Maze -> Node -> [Node]
neighborsOf m n = do
  let xv = x n
  let yv = y n
  let coords = Prelude.filter filterCoords [(xv+1, yv), (xv, yv+1), (xv-1, yv),
(xv, yv-1)]
  filter (\a -> (ttype a) /= Wall) (map (\a -> findNodeByIndex m a) coords)
  where
    filterCoords (a, b) = (a >= 0) && (a < xlenv) && (b >= 0) && (b < ylenv)
    xlenv = xlen m
    ylenv = ylen m

allNeighborsOf :: Maze -> Node -> [Node]
allNeighborsOf m n = do
  let xv = x n
  let yv = y n
  let coords = Prelude.filter filterCoords [(xv+1, yv), (xv, yv+1), (xv-1, yv),
(xv, yv-1)]
  map (\a -> findNodeByIndex m a) coords
  where
    filterCoords (a, b) = (a >= 0) && (a < xlenv) && (b >= 0) && (b < ylenv)
    xlenv = xlen m
    ylenv = ylen m

-- returns the indices in the 2D array of nodes with a given TileType. Used to
find
-- the start and goal nodes.
findIndexByType :: [[Node]] -> TileType -> (Int, Int)
findIndexByType [] _ = (-1, -1)
findIndexByType (r:rs) tt = do
  case (Prelude.filter (\n -> (ttype n) == tt) r) of
    [n] -> (x n, y n)
    _ -> findIndexByType rs tt

-- takes in a row of characters, an xlen, a ylen, and returns the maze.
-- would it be worth parallelizing this for big mazes?
buildMaze :: [[Char]] -> Int -> Int -> Maze
buildMaze mlines ylen xlen = do
  let rows = toRows (zip mlines [0..])
  let start = findIndexByType rows Start
  let goal = findIndexByType rows Goal
  let vecs = Vec.fromList (map (\a -> Vec.fromList a) rows) -- make [[Node]]
into a 2D vector
  Maze { rows = vecs, xlen = xlen, ylen = ylen, start = start, goal = goal }
    where toRows [] = [] -- this returns [[Node]]
          toRows ((l, y):ls) = (map toNode charsWithIndex) : (toRows ls)
            where charsWithIndex = map (\(a, b) -> (a, b, y)) (zip l [0..])
          toNode (n, x, y) = do
            let ttype = case n of
                    ' ' -> Step
                    's' -> Start
                    'g' -> Goal
                    'X' -> Wall
                    _   -> Step -- Shouldn't happen in a valid maze
            Node { ttype = ttype, x = x, y = y }

main :: IO ()
main = do
  args <- getArgs
  case args of
    (filename:depth:xs) -> do
      contents <- readFile filename
      let mlines = lines contents
      let ylen = length mlines
      if (ylen == 0)
        then die ("Maze is invalid: height is 0")
        else return ()
      let xlen = length (head mlines)
      if (xlen == 0)
        then die ("Maze is invalid: length is 0")
        else return ()
      if not (Prelude.foldr (\a b -> b && ((length a) == xlen)) True mlines)
        then die ("Maze is invalid: row lengths vary")
        else return ()
      let ndepth = read depth
      let maze = buildMaze mlines ylen xlen
      let path = aStarPar maze ndepth
      let show_maze = case xs of
            ("-show":[]) -> True
            _ -> False
      putStrLn ((show maze) ++ "\n\n" ++ (showPath maze path show_maze))
    _ -> do
      prog <- getProgName
      die ("Usage: " ++ prog ++ " <maze-filename> <depth> [-show] +RTS
-N<depth>")
```

The small Haskell program I wrote to practice using parList to parallelize evaluating list elements in parallel is as follows, including compilation and running instructions. Note that this program really doesn't compute anything interesting; I just wrote it so I could practice using parList and time the difference it made in evaluating list elements.

```
{-
compile with:
$ stack install monad-par
$  stack  --resolver  lts-22.33  ghc  --  --make  -Wall  -O  partest.hs  -threaded
-rtsopts

to run in serial:
./partest - <depth>
to run in parallel:
./partest + <depth> +RTS -N<depth>

Note that you should still supply the same <depth> even if running in serial,
since the total
amount of work done is based on the specified depth, even if only using one
thread.

This is a test program I wrote to test out evaluating a list of values in
parallel, which
is a strategy I will eventually use in my final program. It does nothing
interesting,
but I used it to time parallel list operations.
-}

import System.Environment
import Control.Parallel.Strategies

testlist :: Int -> [Int]
testlist depth = do
  [x | a <- [0..depth], let x = foldr (+) (0 :: Int) (replicate ((100000000 ::
Int) + a) (1 :: Int))]

main :: IO ()
main = do
  args <- getArgs
  case args of
    [p, depth] -> do
      let numdepth = read depth
      case p of
```

```
        "-" -> do
          let res = testlist numdepth
          putStrLn (show (foldr1 (\a b -> if a > b then a else b) res))
        "+" -> do
          let res = testlist numdepth `using` parList rseq
          putStrLn (show (foldr1 (\a b -> if a > b then a else b) res))
        _ -> putStrLn("usage error 2")
    _ -> putStrLn("usage error 1")
```

The Python3 program I wrote to generate mazes for testing is below, and is also attached as `mazegen.py`. To run the program, you can run "`python3 mazegen.py`" if you have Python3 installed.

```
#! /usr/bin/python3
import sys
import random
import queue

INITIAL = '?'
START = 's'
GOAL = 'g'
WALL = 'X'
PATH = ' '

def get_neighbors(of, xlen, ylen):
    y = of[0]
    x = of[1]
    candidates = [(y, x+1), (y, x-1), (y-1, x), (y+1, x)]
    return filter(lambda a: (a[0] >= 0) and (a[0] < xlen) and (a[1] >= 0) and
(a[1] < ylen), candidates)

def generate_maze(xlen, ylen, likelihood, unreachable_paths):
    maze = [[INITIAL for i in range(xlen)] for j in range(ylen)]
    start = (random.randint(0, xlen-1), random.randint(0, ylen-1))
    q = queue.Queue()
    reachables = []

    maze[start[1]][start[0]] = START
    q.put(start)

    # randomly create paths
    while len(reachables) == 0: # so we don't create a maze with no other
spaces
        while not q.empty():
            cur = q.get()
            ns = get_neighbors(cur, xlen, ylen)
            for n in ns:
                if maze[n[1]][n[0]] == INITIAL:
                    if random.randint(0, 100) <= likelihood:
                        # wall
                        maze[n[1]][n[0]] = WALL
                    else:
                        # pathway
                        maze[n[1]][n[0]] = PATH
                        q.put(n)
                        reachables.append(n)

    # pick a reachable goal
    goal = reachables[random.randint(0, len(reachables) - 1)]
    maze[goal[1]][goal[0]] = GOAL

    # mark unreachable spots
    for y in range(ylen):
        for x in range(xlen):
            if maze[y][x] == INITIAL:
                if (not unreachable_paths) or (random.randint(0, 100) <=
likelihood):
                    maze[y][x] = WALL
                else:
                    maze[y][x] = PATH
    return maze

def print_maze(maze):
    for row in maze:
        rowstr = "".join(row)
        print(rowstr)

if __name__ == "__main__":
    arglen = len(sys.argv)
    if (arglen != 3) and (arglen != 4) and (arglen != 5):
        sys.exit("Usage: mazegen.py <x-len> <y-len> [<wall-likelihood>]
[-no-fill]")

    xlen = int(sys.argv[1])
    ylen = int(sys.argv[2])
    likelihood = int(sys.argv[3]) if (arglen == 4) else 33
    unreachable_paths = (arglen == 5) and (sys.argv[4] == "-no-fill")

    print_maze(generate_maze(xlen, ylen, likelihood, unreachable_paths))
```

References

Mohsin Rizvi
COMS 4995 - Parallel Functional Programming
Fall 2024
Final Report

[1] S. S. Zaghloul, H. Al-Jami, M. Bakalla, L. Al-Jebreen, M. Arshad, and A. Al-Issa, “Parallelizing A* Path Finding Algorithm,” International Journal Of Engineering And Computer Science, vol. 6, no. 9, pp. 22469–22476, Sep. 2017, doi: https://doi.org/10.18535/ijecs/v6i9.13.

Haskell documentation:
System.Environment documentation:
https://hackage.haskell.org/package/base-4.20.0.1/docs/System-Environment.html
System.Exit documentation:
https://hackage.haskell.org/package/haskell2020-0.1.0.0/docs/System-Exit.html
Data.Vector documentation: https://hackage.haskell.org/package/vector-0.13.2.0/docs/Data-Vector.html
Data.PSQueue documentation:
https://hackage.haskell.org/package/PSQueue-1.2.0/docs/Data-PSQueue.html
Data.Map documentation: https://hackage.haskell.org/package/containers-0.4.0.0/docs/Data-Map.html
Data.Maybe documentation: https://hackage.haskell.org/package/base-4.20.0.1/docs/Data-Maybe.html
Data.Set documentation: https://hackage.haskell.org/package/containers-0.7/docs/Data-Set.html
Control.Parallel.Strategies documentation:
https://hackage.haskell.org/package/parallel-3.2.2.0/docs/Control-Parallel-Strategies.html