Parallelizing a Maze Solver using an A * Search Algorithm

Reid Jesselson

Introduction

The first step for the project was to create a sequential maze solving program that utilized an A* search algorithm. The program I wrote takes a file as a command line arguments, reads the maze information from the file, then runs each maze through an A* maze solving function sequentially. The file containing the mazes is formatted such that each line contains a maze. The mazes are formatted with the following rules. The start of the maze is always the top right corner and the solution to the maze is always the bottom right corner. The maze is depicted as a 2D array of 1's and 0's in which 1's represent open space and 0's represent walls.

Obtaining the mazes

The first problem I had to address was finding a suitable data set of mazes to use in my solver, then convert the mazes to the correct format in the file I need. Eventually, I was able to find an open source maze generator in java that outputted the mazes as 2d arrays in a manner similar to what I needed. After some modifications, I was able to change the program to output the mazes in the proper format. This allowed me to generate the necessary mazes of any dimension.

A* Algorithm

As mentioned above, I utilized an A* algorithm to solve the mazes for this project. The steps to the A* algorithm for solving a maze are as follows

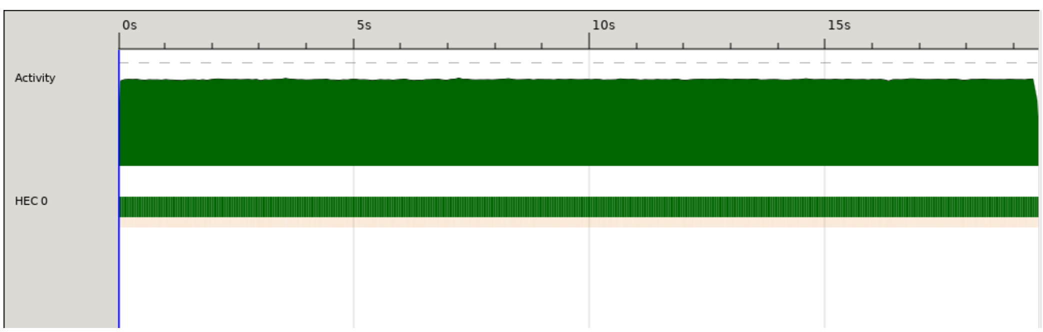
1. Initiate the openlist to contain your starting node and the closedlist to be the empty list
2. Pop the first node of the openlist, call it n
3. Find all neighbors of n that are on the maze and are not walls
4. For each neighbor
    a. If the neighbor is the end node, stop searching and recursively determine the route through the maze
    b. Otherwise, calculate the heuristic, which is equal to the distance traveled so far + the manhattan distance to the end node
    c. If there does not already exists a node on the closedlist or openlist with a lower heuristic value than this node, add it to the open list
5. Put no on the closed list
6. Sort the open list by ascending heuristic value
7. Repeat steps 2 - 6

First Approach

The first approach to parallelizing the maze solved consisted of solving a large number of mazes in parallel. To do this, the parMap function was used to iterate through the list of mazes to be solved, generating a spark for each maze. The results comparing the sequential maze solver to this iteration of the parallel maze solver are shown below:

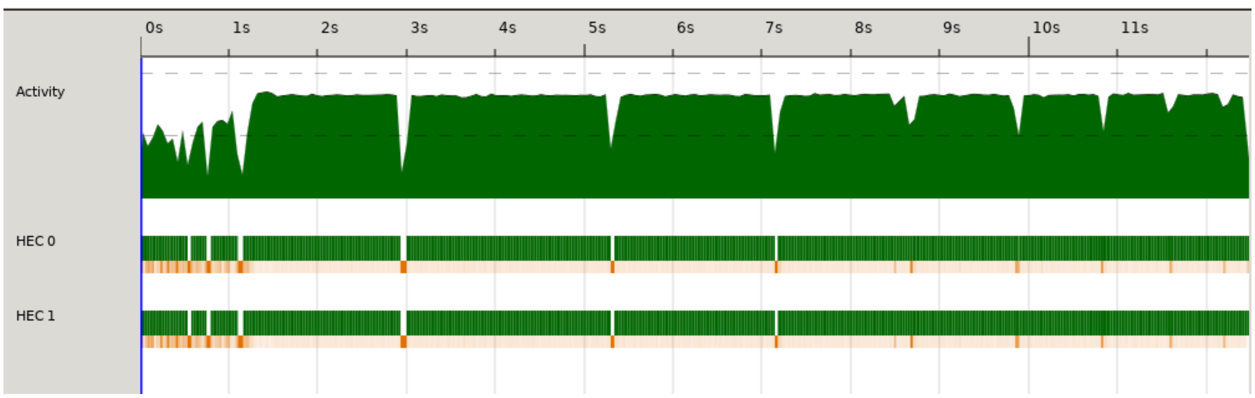
When tested on solving 500 100x100 mazes, the sequential maze solver took ~19 seconds to solve all 500 mazes. The output and threadscope can be viewed below.

```
110,251,273,640 bytes allocated in the heap
  6,226,340,368 bytes copied during GC
      1,030,744 bytes maximum residency (1383 sample(s))
         40,368 bytes maximum slop
              4 MiB total memory in use (0 MB lost due to fragmentation)

                                   Tot time (elapsed)  Avg pause  Max pause
 Gen  0     104050 colls,     0 par    2.016s   2.241s     0.0000s    0.0004s
 Gen  1       1383 colls,     0 par    0.344s   0.344s     0.0002s    0.0006s

 INIT    time    0.000s  (  0.000s elapsed)
 MUT     time   16.500s  ( 16.384s elapsed)
 GC      time    2.359s  (  2.585s elapsed)
 EXIT    time    0.000s  (  0.000s elapsed)
 Total   time   18.859s  ( 18.969s elapsed)

 %GC     time       0.0%  (0.0% elapsed)

 Alloc rate    6,681,895,372 bytes per MUT second

 Productivity  87.5% of total user, 86.4% of total elapsed
```



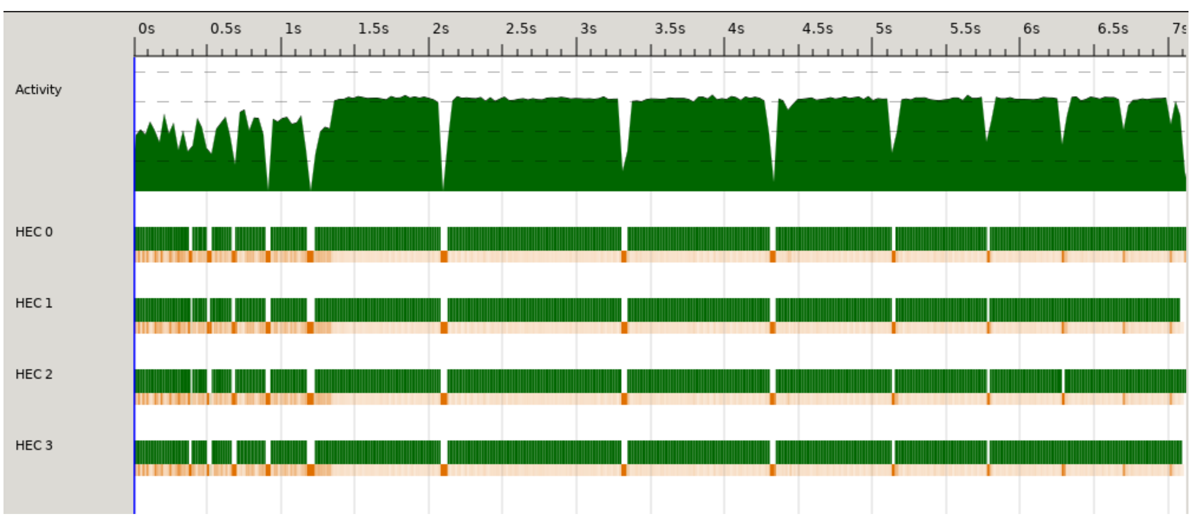
When tested on solving 500 100x100 mazes, the parallel maze solver utilizing 2 cores took ~12.5 seconds to solve all 500 mazes. This gives a speedup of 19/12.5 ~= 1.52. The output and threadscope can be viewed below. 500 sparks in total were generated (1 for each of the 500 mazes) and each spark was converted.

```
110,435,510,960 bytes allocated in the heap
  7,543,918,968 bytes copied during GC
    223,724,792 bytes maximum residency (27 sample(s))
      1,407,752 bytes maximum slop
            621 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
 Gen  0     62147 colls, 62147 par    3.344s   2.031s     0.0000s    0.0004s
 Gen  1        27 colls,    26 par    0.875s   0.524s     0.0194s    0.0641s

 Parallel GC work balance: 70.89% (serial 0%, perfect 100%)

 TASKS: 4 (1 bound, 3 peak workers (3 total), using -N2)

 SPARKS: 500 (500 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

 INIT    time    0.000s  (  0.000s elapsed)
 MUT     time   19.750s  (  9.999s elapsed)
 GC      time    4.219s  (  2.555s elapsed)
 EXIT    time    0.000s  (  0.000s elapsed)
 Total   time   23.969s  ( 12.555s elapsed)

 Alloc rate    5,591,671,441 bytes per MUT second

 Productivity  82.4% of total user, 79.6% of total elapsed
```



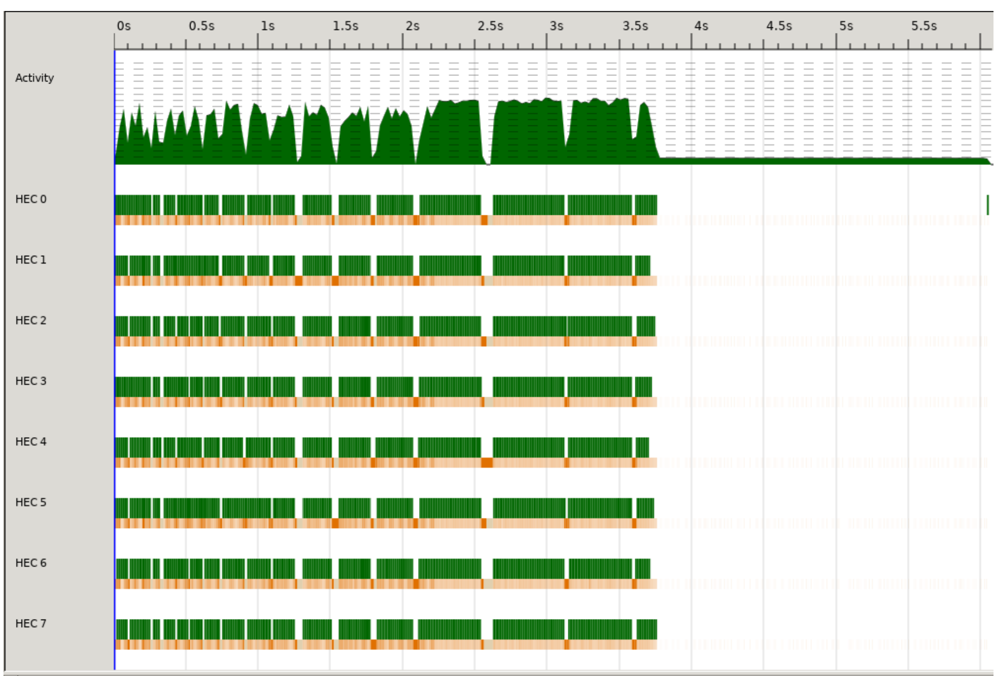
When tested on solving 500 100x100 mazes, the parallel maze solver utilizing 4 cores took ~7.1 seconds to solve all 500 mazes. This gives a speedup of 19/7.1 ~= 2.67. The output and threadscope can be viewed below. Again, 500 sparks in total were generated and each spark was converted.

```
110,430,590,616 bytes allocated in the heap
  7,808,509,424 bytes copied during GC
    204,682,040 bytes maximum residency (27 sample(s))
      2,309,320 bytes maximum slop
            586 MiB total memory in use (0 MB lost due to fragmentation)

                                   Tot time (elapsed)  Avg pause  Max pause
 Gen  0     33712 colls, 33712 par    4.250s   1.493s     0.0000s    0.0005s
 Gen  1        27 colls,    26 par    1.359s   0.462s     0.0171s    0.0548s

 Parallel GC work balance: 73.86% (serial 0%, perfect 100%)

 TASKS: 6 (1 bound, 5 peak workers (5 total), using -N4)

 SPARKS: 500 (500 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

 INIT    time    0.000s  (  0.000s elapsed)
 MUT     time   21.141s  (  5.143s elapsed)
 GC      time    5.609s  (  1.955s elapsed)
 EXIT    time    0.000s  (  0.000s elapsed)
 Total   time   26.750s  (  7.098s elapsed)

 Alloc rate    5,223,619,955 bytes per MUT second

 Productivity  79.0% of total user, 72.5% of total elapsed
```



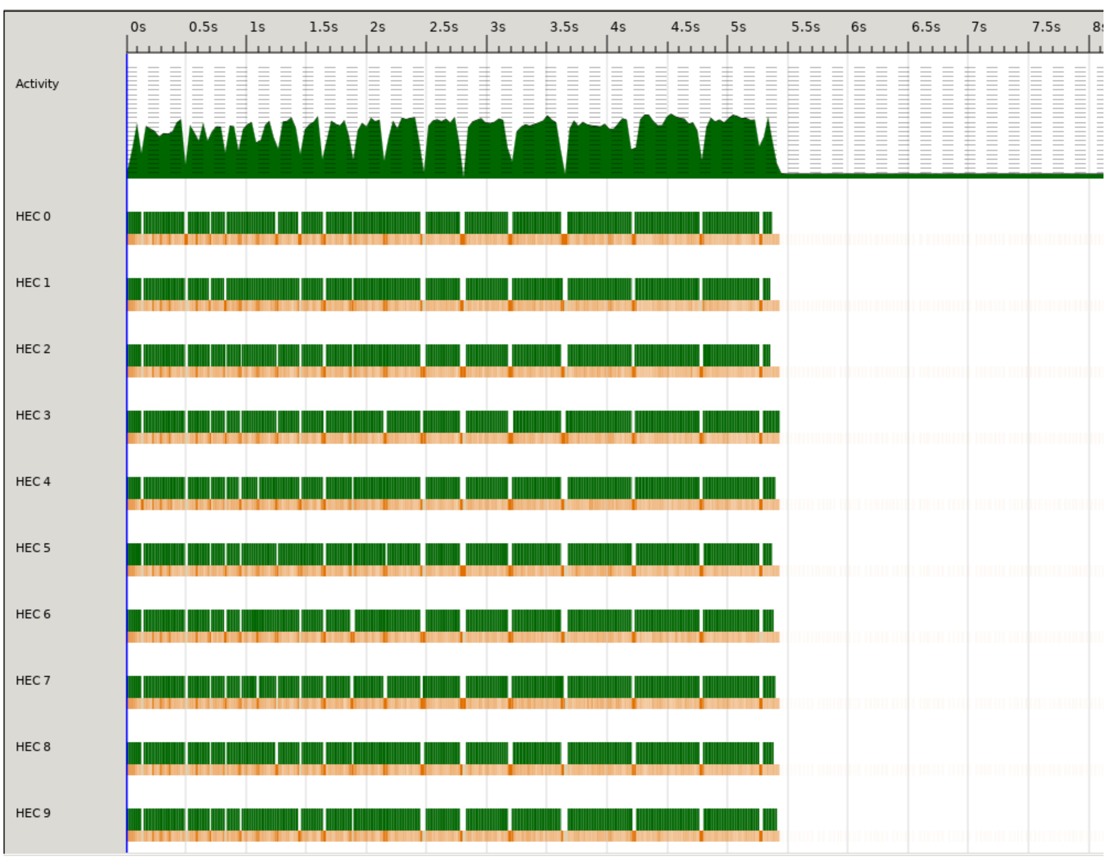
When tested on solving 500 100x100 mazes, the parallel maze solver utilizing 16 cores took ~6.1 seconds to solve all 500 mazes. This gives a speedup of 19/6.1 ~= 3.11. The output and threadscope can be viewed below. Again, 500 sparks in total were generated and each spark was converted.

```
110,429,105,848 bytes allocated in the heap
  9,677,068,992 bytes copied during GC
    184,503,232 bytes maximum residency (29 sample(s))
      8,647,752 bytes maximum slop
            555 MiB total memory in use (0 MB lost due to fragmentation)

                                    Tot time (elapsed)  Avg pause  Max pause
 Gen  0     10906 colls, 10906 par   14.547s   1.271s     0.0001s    0.0007s
 Gen  1        29 colls,    28 par    8.188s   0.641s     0.0221s    0.0550s

 Parallel GC work balance: 79.89% (serial 0%, perfect 100%)

 TASKS: 18 (1 bound, 17 peak workers (17 total), using -N16)

 SPARKS: 500 (500 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

 INIT    time    0.000s  (  0.001s elapsed)
 MUT     time   30.156s  (  4.248s elapsed)
 GC      time   22.734s  (  1.912s elapsed)
 EXIT    time    0.000s  (  0.000s elapsed)
 Total   time   52.891s  (  6.161s elapsed)

 Alloc rate    3,661,897,810 bytes per MUT second

 Productivity  57.0% of total user, 69.0% of total elapsed
```



When tested on solving 500 100x100 mazes, the parallel maze solver utilizing 24 cores took ~6.7 seconds to solve all 500 mazes. This gives a speedup of 19/7.2 ~= 2.83. The output and threadscope can be viewed below. Again, 500 sparks in total were generated and each spark was converted.

```
110,433,513,992 bytes allocated in the heap
  9,394,379,296 bytes copied during GC
    177,057,864 bytes maximum residency (32 sample(s))
     10,299,704 bytes maximum slop
            534 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
  Gen  0      6740 colls,  6740 par   17.297s   1.234s     0.0002s    0.0007s
  Gen  1        32 colls,    31 par   12.188s   0.659s     0.0206s    0.0779s

  Parallel GC work balance: 80.91% (serial 0%, perfect 100%)

  TASKS: 28 (1 bound, 27 peak workers (27 total), using -N24)

  SPARKS: 500 (500 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)

  INIT    time    0.000s  (  0.001s elapsed)
  MUT     time   51.406s  (  4.770s elapsed)
  GC      time   29.484s  (  1.893s elapsed)
  EXIT    time    0.078s  (  0.000s elapsed)
  Total   time   80.969s  (  6.665s elapsed)

  Alloc rate    2,148,250,728 bytes per MUT second

  Productivity  63.5% of total user, 71.6% of total elapsed
```



Significant speedups occurred in the parallel implementation of the maze solver for each number of cores used. The full chart can be found below.

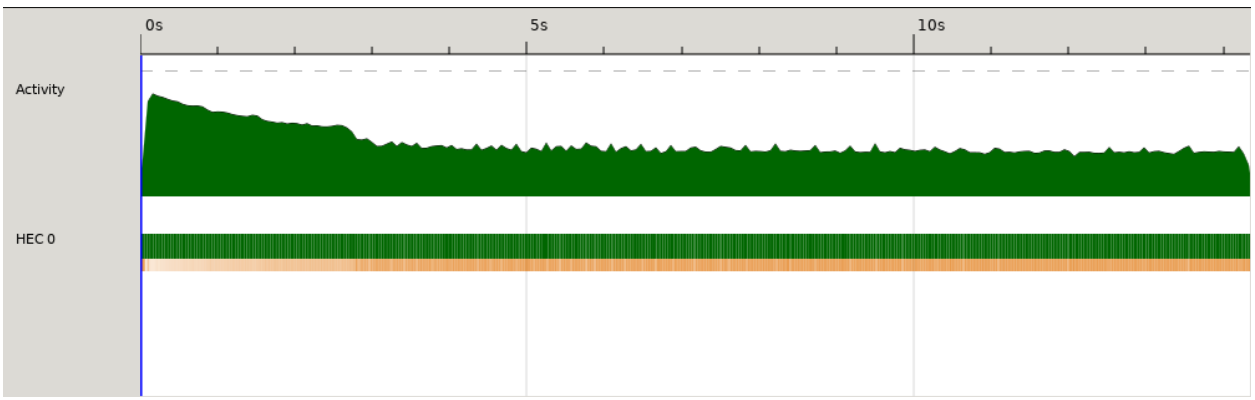
# of Cores	Runtime (Speed Up)
Sequential (1 Core)	19 seconds (1)
Parallel (2 Cores)	12.5 seconds (1.52)
Parallel (4 Cores)	7.1 seconds (2.67)
Parallel (16 Cores)	6.1 seconds (3.11)
Parallel (24 Cores)	6.7 seconds (2.83)

Next Approach

Now that I had parallelized the solving of multiple mazes, I wanted to parallelize my algorithm for solving an individual maze of a large size. A* is inherently a somewhat difficult algorithm to parallelize, but there are a few different methods to consider. The first approach I chose was the evaluation of the heuristic. In the A* search algorithm, the calculation of the heuristic is typically a very time consuming step. Because of this, being able to calculate the heuristic of multiple nodes in parallel should yield a significant decrease in runtime. To do this, once the neighbor nodes had been calculated for the node on the front of the open list in our A* algorithm, instead of sequentially calculating the heuristic for each node, parMap was used to calculate the heuristic for each node in parallel. The results comparing this implementation of calculating the heuristic in parallel vs sequential are shown below.

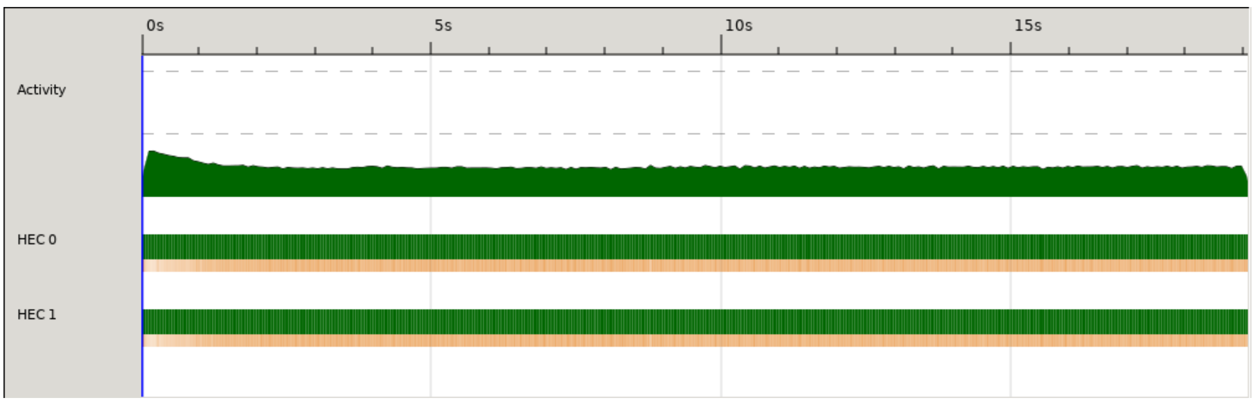
When tested on solving a 500x500 maze, the sequential maze solver took ~14.5 seconds to solve all 500 mazes. The output and threadscope can be viewed below.

```
33,318,995,200 bytes allocated in the heap
23,347,110,304 bytes copied during GC
    23,556,792 bytes maximum residency (385 sample(s))
       238,112 bytes maximum slop
            71 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
Gen  0     31456 colls,     0 par    4.859s   5.062s     0.0002s    0.0006s
Gen  1       385 colls,     0 par    3.547s   3.193s     0.0083s    0.0167s

INIT    time    0.000s  (  0.000s elapsed)
MUT     time    6.125s  (  6.349s elapsed)
GC      time    8.406s  (  8.255s elapsed)
EXIT    time    0.000s  (  0.000s elapsed)
Total   time   14.531s  ( 14.604s elapsed)

%GC     time       0.0%  (0.0% elapsed)

Alloc rate    5,439,835,951 bytes per MUT second

Productivity  42.2% of total user, 43.5% of total elapsed
```



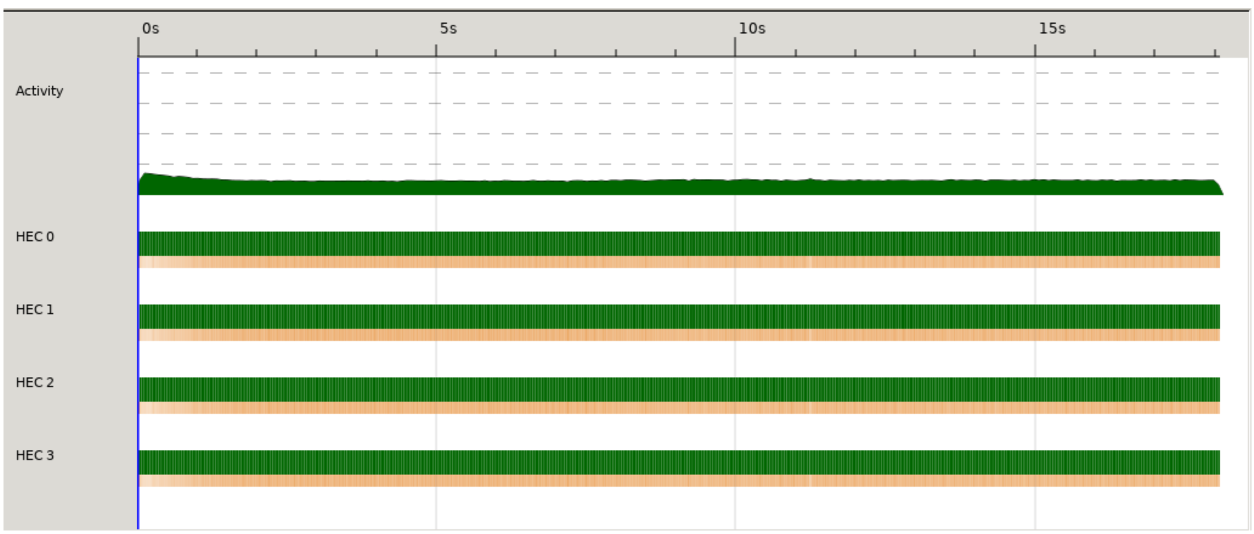
When tested on solving a 500x500 maze, the parallel maze solver utilizing 2 cores took ~18.3 seconds to solve all 500 mazes. This gives a speedup of 14.5/18.3 = .79, showing that this parallel implementation leads to an increase in runtime. We will analyze that further later on in the report. The output and threadscope can be viewed below. In total 68,385 sparks were generated. Of those 28603 were converted, 39686 were GC'd and 96 fizzled.

```
70,774,975,736 bytes allocated in the heap
30,191,028,528 bytes copied during GC
    23,558,168 bytes maximum residency (608 sample(s))
       374,008 bytes maximum slop
            72 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
Gen  0     67444 colls, 67444 par    9.094s   6.255s     0.0001s    0.0004s
Gen  1       608 colls,   607 par    5.547s   2.772s     0.0046s    0.0074s

Parallel GC work balance: 41.33% (serial 0%, perfect 100%)

TASKS: 4 (1 bound, 3 peak workers (3 total), using -N2)

SPARKS: 68385 (28603 converted, 0 overflowed, 0 dud, 39686 GC'd, 96 fizzled)

INIT    time    0.000s  (  0.000s elapsed)
MUT     time    9.219s  (  9.296s elapsed)
GC      time   14.641s  (  9.027s elapsed)
EXIT    time    0.000s  (  0.000s elapsed)
Total   time   23.859s  ( 18.324s elapsed)

Alloc rate    7,677,285,503 bytes per MUT second

Productivity  38.6% of total user, 50.7% of total elapsed
```



When tested on solving a 500x500 maze, the parallel maze solver utilizing 4 cores took ~18.1 seconds to solve all 500 mazes. This gives a speedup of 14.5/18.1 = .81. This is slightly faster than the implementation with 2 cores, but still significantly slower than the sequential implementation. The output and threadscope can be viewed below. In total 68,385 sparks were generated. Of those 28614 were converted, 39693 were GC'd and 78 fizzled. The total number of sparks, and the result is very similar to the 2 core implementation.

```
70,829,024,936 bytes allocated in the heap
30,255,680,024 bytes copied during GC
    23,558,904 bytes maximum residency (611 sample(s))
       442,328 bytes maximum slop
            74 MiB total memory in use (0 MB lost due to fragmentation)

                                     Tot time (elapsed)  Avg pause  Max pause
Gen  0     67441 colls, 67441 par   13.938s   6.322s     0.0001s    0.0011s
Gen  1       611 colls,   610 par    7.922s   2.163s     0.0035s    0.0058s

Parallel GC work balance: 40.88% (serial 0%, perfect 100%)

TASKS: 6 (1 bound, 5 peak workers (5 total), using -N4)

SPARKS: 68385 (28614 converted, 0 overflowed, 0 dud, 39693 GC'd, 78 fizzled)

INIT    time    0.000s  (  0.000s elapsed)
MUT     time   10.578s  (  9.583s elapsed)
GC      time   21.859s  (  8.485s elapsed)
EXIT    time    0.000s  (  0.000s elapsed)
Total   time   32.438s  ( 18.068s elapsed)

Alloc rate    6,695,801,471 bytes per MUT second

Productivity  32.6% of total user, 53.0% of total elapsed
```



We did not find speed ups in any of the parallel implementations of the maze solver for each number of cores used. The full chart can be found below.

# of Cores	Runtime (Speed up)
Sequential (1)	14.5 (1)
Parallel (2)	18.3 (.79)
Parallel (4)	18 (.81)

Conclusions

For solving a large number of mazes in parallel my algorithm worked very well. The speedup factor increased with the number of cores before steady leveling off at just above a factor of 3. A speedup factor of 3.11 for 16 cores indicates that my algorithm for parallelizing the solving of mazes is effective.

For solving a large, individual maze my algorithm did not work as I had hoped. From the data above, we can see that parallelizing the calculation of the heuristic for the nodes being expanded did not lead to the desired decrease in run time. In fact, it actually led to a slight increase in runtime for both the 2 and 4 core implementations. This is likely due to the fact that the inherent overhead required by using parMap is greater than the benefits of using a parallel implementation to calculate the heuristic. The reason this is the case is because I was only able to calculate at most 4 (for the 4 neighboring cells) heuristics concurrently, thus we were only running parMap on lists of length 4 or less. I was unable to discover a way to efficiently calculate all of the heuristics at once in parallel, then be able to access them when necessary throughout the algorithm. Overall, this aspect of the project did not go the way I had hoped.

There is certainly opportunity to speedup the individual maze solving problem. One possibility I considered was using a Divide and Conquer approach, where a larger maze is divided in several smaller mazes. These smaller mazes are solved in parallel, then the individual solutions are combined to find an overall solution to the larger maze. However, I was unable to determine how to best select the (start,end) pairs for each of the smaller mazes to ensure that a solution to the maze would be found should one exist. Another possibility I investigated was parallelizing the selection of the nodes from the openlist in A*. (I.E. expanding on multiple nodes in parallel) But my implementation of this led to a very significant increase in runtime.

Code Listing Part 1(Solving Multiple Mazes in Parallel)

```
import Control.Exception
import System.Environment
import Data.Maybe
import Data.List (sortBy)
import Data.Ord (comparing)
import Data.Char (digitToInt)
import Data.List.Split (splitOn)
import Control.Parallel.Strategies hiding (parMap)
import Control.Seq as Seq
import Control.DeepSeq

type Maze = [[Int]]
type Coord = (Int, Int)
type Route = [Coord]
data Node = Node { pos :: Coord,
                   d :: Int,
                   f :: Int,
                   parent :: Coord
                   } deriving (Eq, Show)

getValidNeighbors :: Coord -> Maze -> [Coord]
getValidNeighbors (x, y) maze = filter (isOpen maze) [(x1, y1) | (x1,y1)
<- [(x+1,y),(x-1,y),(x,y+1),(x,y-1)], x1 < (length maze),  x1 >= 0, y1 <
(length (maze !! x)), y1 >= 0]

distance :: Coord -> Coord -> Int
distance (x1,y1) (x2,y2) = abs (x1 - x2) + abs(y1 - y2)

isOpen :: Maze -> Coord -> Bool
isOpen maze (x,y) = maze !! x !! y == 1

nodefilter :: Node -> [Node] -> Bool
nodefilter node nodes = any (\x -> pos x == pos node && f x <= f node)
nodes

sortByF :: [Node] -> [Node]
sortByF = sortBy (comparing f)
```

```
buildRoute :: Node -> Coord -> [Node] -> Route
buildRoute cur start nodes
        | pos cur == start = [start]
        | otherwise =  (pos cur) : buildRoute (findNode (parent cur)
nodes) start nodes

findNode :: Coord -> [Node] -> Node
findNode cur (x:xs)
        | cur == pos x = x
        | otherwise = findNode cur xs

start :: Coord
start = (0,0)

solveMaze :: Maze -> Coord -> Coord -> [Node] -> [Node] -> Maybe Route
solveMaze maze cur end openList closedList
        | openList == [] = Nothing
        | cur == end = Just (reverse (buildRoute (head openList) start
closedList))
        | otherwise =
          let
                curNode = head openList
                neighbors = getValidNeighbors cur maze
                neighborNodes = filter (not . checkClosed) [Node
{pos=x,d=(y+1),f=(y+1+(distance x end)),parent=cur} | x <- neighbors, let
y = d curNode]
                        where checkClosed n = nodefilter n (closedList ++
openList)
                curOpenList = sortByF (tail openList ++ neighborNodes)

          in solveMaze maze (pos (head curOpenList)) end curOpenList
(closedList ++ [(head openList)])

main :: IO()
main = do
        [filename] <-getArgs
        contents <- readFile filename
        let mazeLines = lines contents
```

```
        let mazeStrings = (map ( map (splitOn ",")) (map (splitOn " ")
mazeLines))
        let mazes = map (map (map (\x -> read x :: Int))) mazeStrings

        print (length (filter isJust ( deep $ runEval $ parMap solve
mazes)))
        where solve maze = solveMaze maze start ((length maze)-1,(length
(maze !! 0))-1) [Node {pos=start,d=0,f=0,parent=start}] []

parMap :: (a -> b) -> [a] -> Eval [b]
parMap f [] = return []
parMap f (a:as) = do
   b <- rpar (f a)
   bs <- parMap f as
   return (b:bs)

deep :: NFData a => a -> a
deep a = deepseq a a
```

Code List Part 2 (Solving an Individual Maze)

```
import Control.Exception
import System.Environment
import Data.Maybe
import Data.List (sortBy)
import Data.Ord (comparing)
import Data.Char (digitToInt)
import Data.List.Split (splitOn)
import Control.Parallel.Strategies (parMap, rseq)

type Maze = [[Int]]
type Coord = (Int, Int)
type Route = [Coord]
data Node = Node { pos :: Coord,
                  d :: Int,
                  f :: Int,
                  parent :: Coord
                  } deriving (Eq, Show)
```

```
getValidNeighbors :: Coord -> Maze -> [Coord]
getValidNeighbors (x, y) maze = filter (isOpen maze) [(x1, y1) | (x1,y1)
<- [(x+1,y),(x-1,y),(x,y+1),(x,y-1)], x1 < (length maze),  x1 >= 0, y1 <
(length (maze !! x)), y1 >= 0]

distance :: Coord -> Coord -> Int
distance (x1,y1) (x2,y2) = abs (x1 - x2) + abs(y1 - y2)

isOpen :: Maze -> Coord -> Bool
isOpen maze (x,y) = maze !! x !! y == 1

nodefilter :: Node -> [Node] -> Bool
nodefilter node nodes = any (\x -> pos x == pos node && f x <= f node)
nodes

sortByF :: [Node] -> [Node]
sortByF = sortBy (comparing f)

buildRoute :: Node -> Coord -> [Node] -> Route
buildRoute cur start nodes
        | pos cur == start = [start]
        | otherwise =  (pos cur) : buildRoute (findNode (parent cur)
nodes) start nodes

findNode :: Coord -> [Node] -> Node
findNode cur (x:xs)
        | cur == pos x = x
        | otherwise = findNode cur xs

start :: Coord
start = (0,0)

solveMaze :: Maze -> Coord -> Coord -> [Node] -> [Node] -> Maybe Route
solveMaze maze cur end openList closedList
        | openList == [] = Nothing
        | cur == end = Just (reverse (buildRoute (head openList) start
closedList))
        | otherwise =
          let
                curNode = head openList
```

```
                neighbors = getValidNeighbors cur maze

                neighborNodes = parMap rseq getNeighborNodes neighbors
                        where getNeighborNodes x = Node {pos=x,d=((d
curNode) + 1),f=(((d curNode) + 1) + (distance x end)),parent=cur}
                filt =  filter (not . checkClosed) neighborNodes
                        where checkClosed n = nodefilter n (closedList ++
openList)

                curOpenList = sortByF (tail openList ++ filt)

          in solveMaze maze (pos (head curOpenList)) end curOpenList
(closedList ++ [(head openList)])

main :: IO()
main = do
    args <- getArgs
    case args of
        [filename] -> do
                contents <- readFile filename
                let mazeLines = lines contents
                let mazeStrings = (map ( map (splitOn ",")) (map (splitOn
" ") mazeLines))
                let mazes = map (map (map (\x -> read x :: Int)))
mazeStrings
                print (length (filter isJust (map solve mazes)))
                        where solve maze = solveMaze maze start ((length
maze)-1,(length (maze !! 0))-1) [Node {pos=start,d=0,f=0,parent=start}] []
```