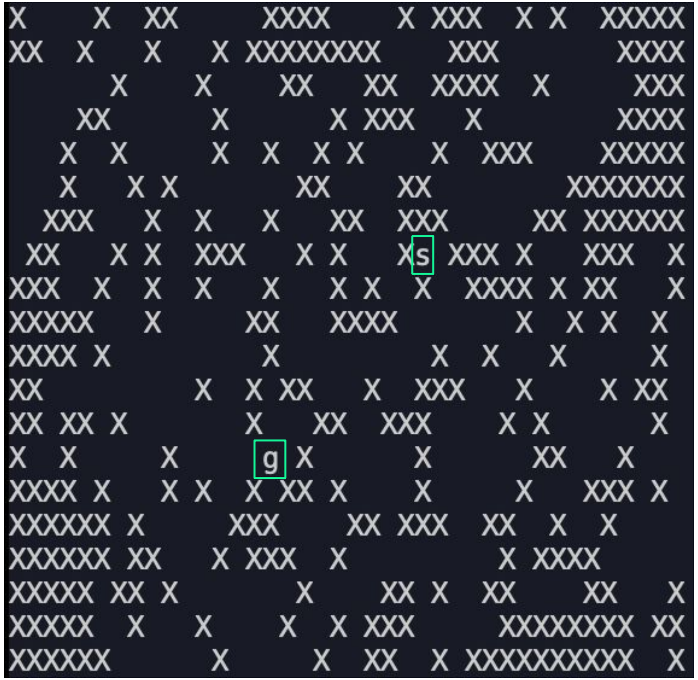
Parallel Maze Solver

Solving mazes in parallel with A*
Mohsin Rizvi
COMS 4995 Parallel Functional Programming

The problem

- Given a grid-based maze, find the shortest path from a known start to a known goal
- Mazes are represented as a series of **tiles**, where some tiles are impassable (“walls”)
- Each maze tile is identifiable by its coordinates
- A path is a list of tiles to move to, from the start tile to the goal tile



```
X    X  XX     XXXX    X XXX  X X  XXXXX
XX  X   X   X XXXXXXXX    XXX       XXXX
      X    X    XX   XX  XXXX  X     XXX
    XX      X      X XXX   X        XXXX
   X  X     X  X  X X    X  XXX    XXXXX
   X   X X       XX    XX        XXXXXXX
  XXX   X  X   X   XX  XXX     XX XXXXXX
 XX    X X  XXX   X X   Xs XXX X   XXX  X
XXX  X  X  X   X   X X  X  XXXX X XX   X
XXXXX   X     XX   XXXX       X  X X  X
XXXX X         X         X  X   X     X
XX         X  X XX   X  XXX   X    X XX
XX XX X       X   XX  XXX    X X      X
X  X     X     g X      X      XX   X
XXXX X   X X  X XX X    X     X   XXX X
XXXXXX X     XXX    XX XXX  XX  X  X
XXXXXX XX   X XXX  X         X XXXX
XXXXX XX X       X    XX X  XX    XX   X
XXXXX  X   X    X  X XXX     XXXXXXXX XX
XXXXXX      X     X  XX  X XXXXXXXXXX  X
```

The A* algorithm

- A* (or A-star) is a generic pathfinding algorithm for finding a path from one weighted graph node to another
- Various applications, including **video games**, **network routing**, and **robotics**
- To use A*, we can think of a grid-based maze as a dense graph
    - All edges have weight 1

The A* algorithm

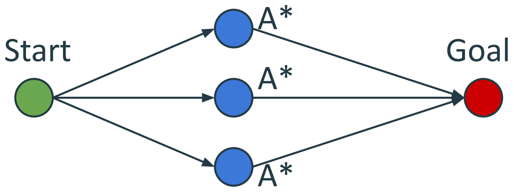
- Relies on a **heuristic function** to estimate a node’s distance to the goal
  - For grid-based mazes, we can use the Euclidean distance to the goal
- Performs a graph search from the start node, adding adjacent nodes to a priority queue
  - Priority is a node’s heuristic value plus the node’s shortest known distance from the start
- Nodes are processed from the priority queue until we find the goal or run out of nodes to search

Parallelization

- Finding the shortest path is **hard** to do fast with parallelization
- You don’t know that a route is the shortest one until you’ve inspected all the alternatives
- Especially difficult if threads don’t have access to a shared priority queue
- I tried two strategies for parallelization, each with their own tradeoffs

Strategy 1: multiple starts

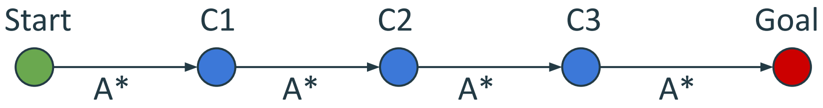
- Launch several A* searches from different points at a fixed distance from the start tile
- Take the shortest result from all the searches
- Inspired by existing literature [1]
- Results:
    - The good: Returned an optimal path
    - The bad: slower than a serial search
        - Each thread still did a full search, so nothing gets sped up



[1] https://www.semanticscholar.org/paper/Parallelizing-A*-Path-Finding-Algorithm-Zaghloul-Al-Jami/8c62a239505647143e3f04fb20d9e5a748a5e47d

Strategy 2: checkpoint partitioning

- This idea came from thinking of how to keep each processor from doing a full search
- The idea: first, come up with “checkpoints” along the ideal path between the start and goal, as if there were no walls in the maze
  - Easy to compute because we have the coordinates of the start and goal
- Next, have each thread compute the path between two checkpoints using a regular A* search
  - Easy to do using `parList` with `rseq`
- When threads are finished, stitch together the resulting paths



Strategy 2: checkpoint partitioning

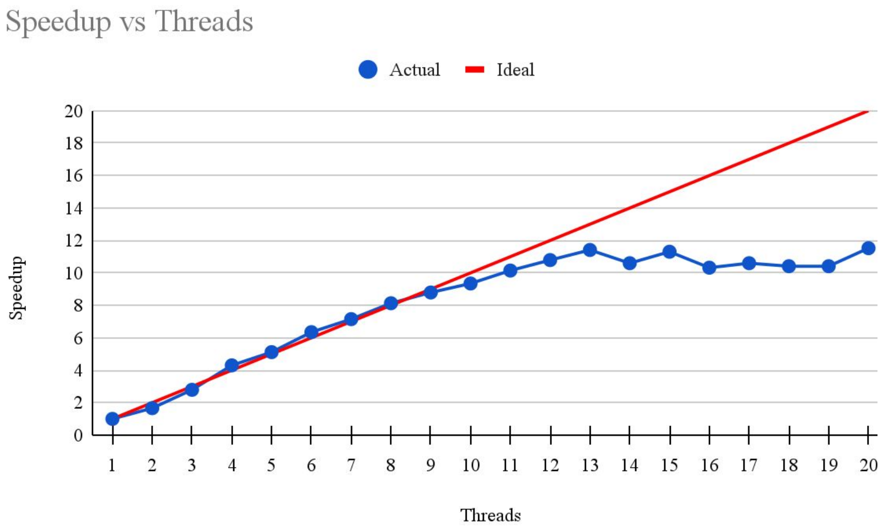
- Results:
  - The good: much faster than serial (more on performance soon)
    - Each thread only did a portion of the full search
  - The bad: paths were *slightly* longer than optimal
    - Sometimes took unnecessary detours to reach checkpoints
    - If a checkpoint isn't reachable from the start or goal, it fails to return any path

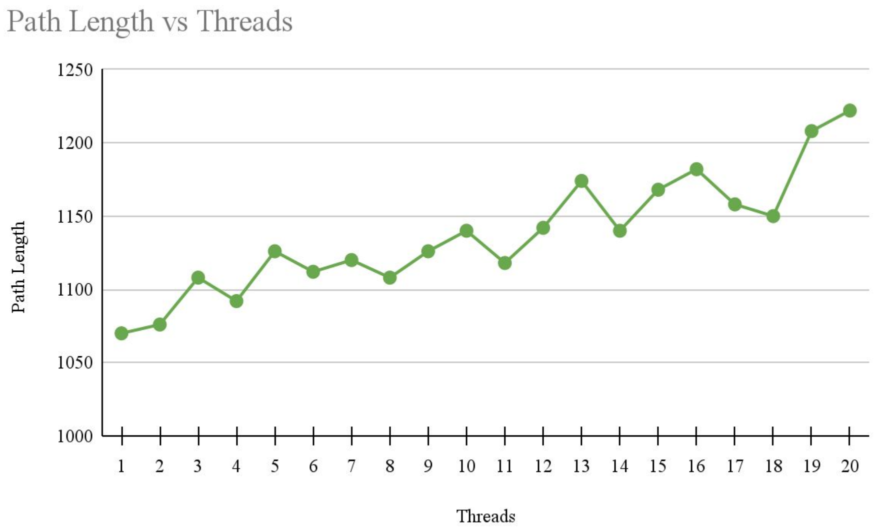
Strategy 2: checkpoint partitioning

- I was able to reduce the path length from detours with **post-processing**
    - If a tile appeared twice in the final path, remove all tiles between the two occurrences
- Tradeoff of this approach: time to compute vs path length
    - This method is suitable if you'd rather compute paths quickly than get an optimal path
    - Also doesn't work if there are unreachable parts of the maze
    - Overall, **speed improvement** was proportionally much greater than the **increase in path length**
        - Resulted in an almost optimal path

Parallel performance

- I chose to use strategy 2 (**checkpoint partitioning**) because of its speed
- On a 200x1000 tile map using up to 20 cores:


Speedup vs Threads

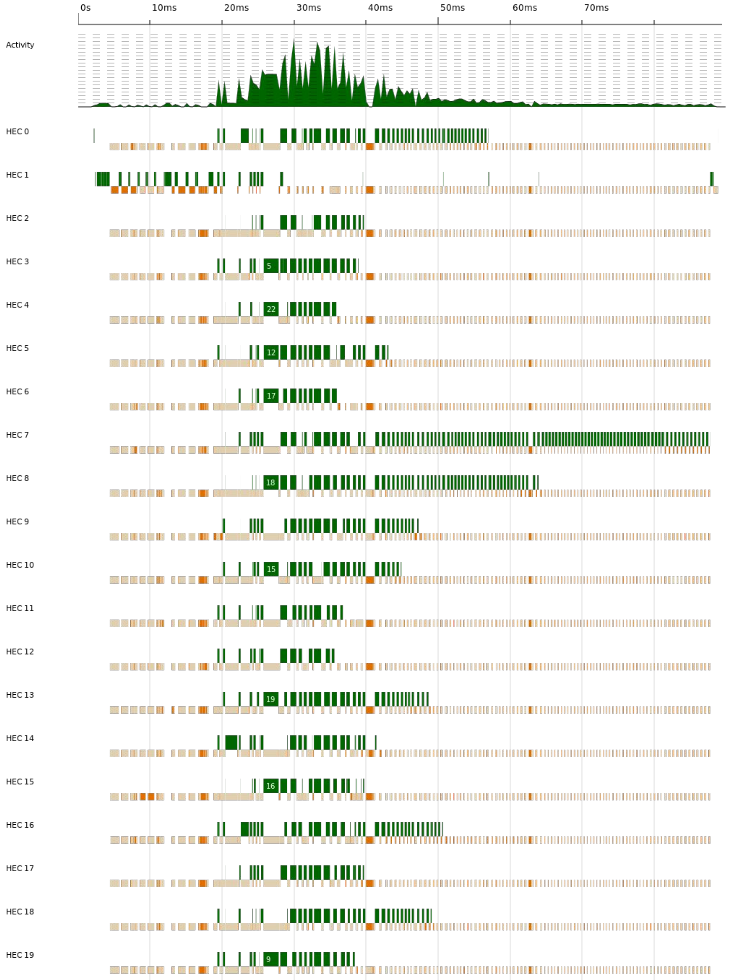

Path Length vs Threads

Parallel performance

- Processor utilization was very good *to a point*
  - Using 8 cores led to about an **8 times speedup**
  - Speed stopped increasing after about 12 cores
- Speed improvement far outweighed path length increase
  - For 12 cores, observed a **10.7 times speedup** and only **6% increase in path length**
  - Reasonable for use in applications that need to compute a lot of paths

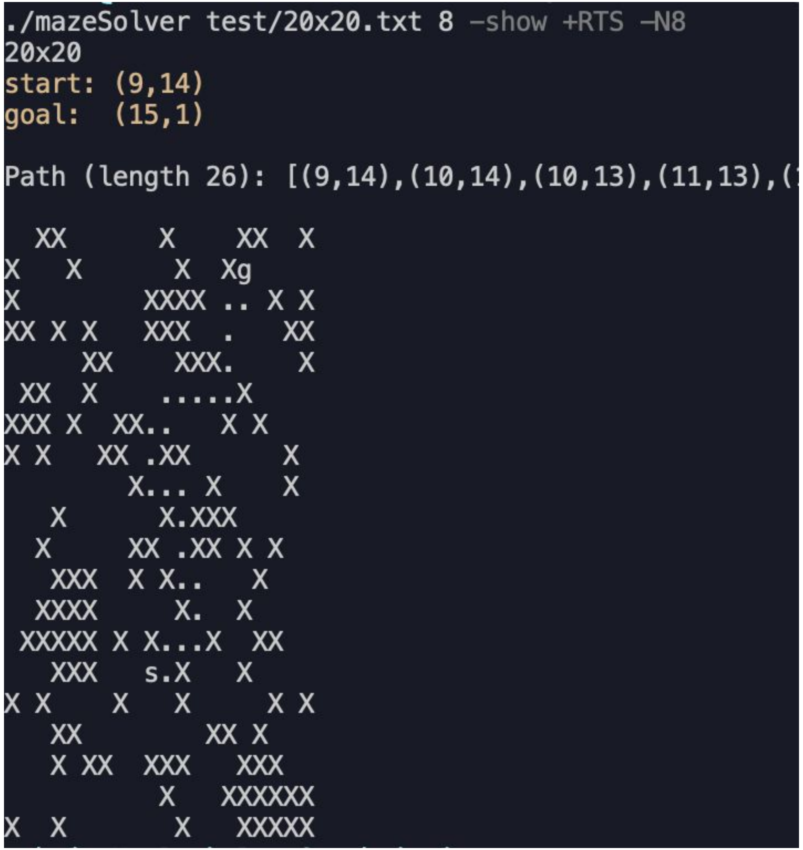
Processor activity

- Workload was **not** evenly distributed amongst processors
- Most processors ended up waiting on one or two more to finish searching
- With this approach, **work distribution is highly dependent on maze layout**
- If we could ensure even work, overall speed would likely increase



The final program

- Takes in a path to a file containing a maze and a level of parallelism to use
- Can optionally render the final path over the maze using the `-show` option
- For example, `./mazeSolver test/20x20.txt 8 -show +RTS -N8`



```
./mazeSolver test/20x20.txt 8 -show +RTS -N8
20x20
start: (9,14)
goal:  (15,1)

Path (length 26): [(9,14),(10,14),(10,13),(11,13),(

  XX      X    XX  X
X   X      X  Xg
X        XXXX .. X X
XX X X   XXX  .   XX
     XX    XXX.    X
 XX  X    .....X
XXX X  XX..   X X
X X   XX .XX      X
        X... X    X
   X      X.XXX
  X     XX .XX X X
   XXX  X X..   X
  XXXX     X.  X
 XXXXX X X...X  XX
   XXX   s.X   X
X X    X   X     X X
   XX        XX X
   X XX  XXX   XXX
          X   XXXXXX
X  X       X   XXXXX
```