IIT DELHI

COL 380 ASSIGNMENT-3 REPORT

MAZE GENERATION AND SOLVING

AUTHORS

UTKARSH SHARMA

Student No. 2021CS10098

RAJAT GOLECHHA

Student No. 2021CS10082

HARDIK GARG

Student No. 2021CS10560

SUPERVISOR

RIJUREKHA SEN

MAY 2024

Contents

1 Introduction

Problem Statement: Creating and solving random maze with graph algorithms on distributed memory

2

Maze Generation

2.1 Resolution Scaling

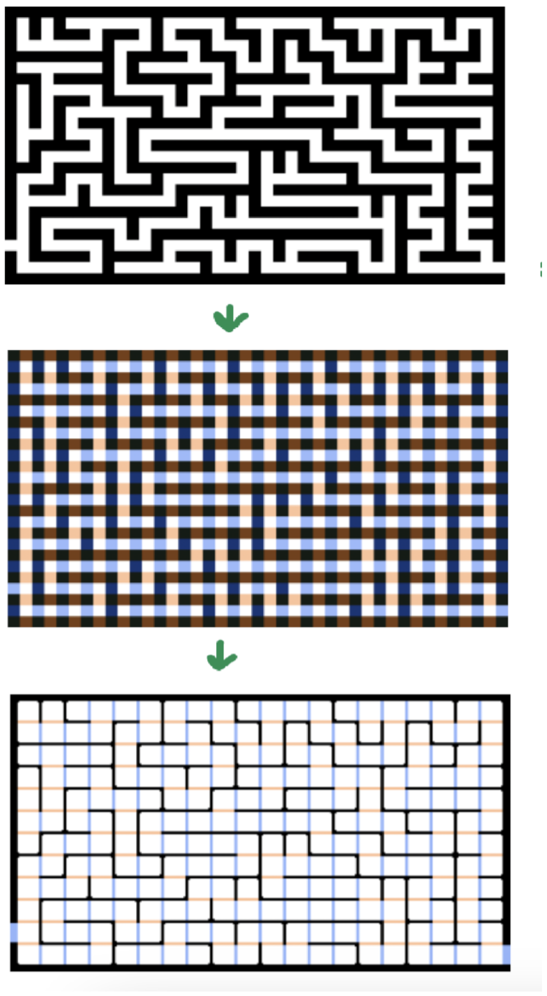
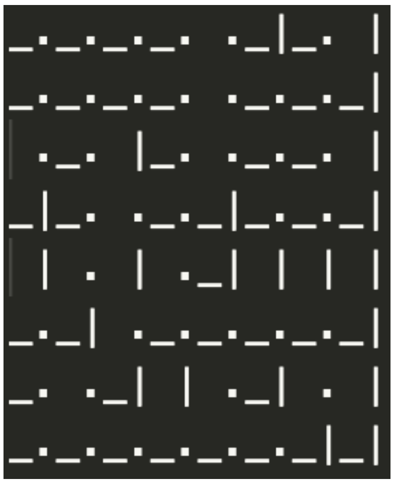


Figure 2.1: *Colour each even row red and even column blue. The diagram shows the equivalence between a maze with thick walls resolution = R, and a maze with infinitely thin walls (resolution = 0.5R)*

Observation - We need a perfect maze such i.e. there exists a unique path between any two vertices

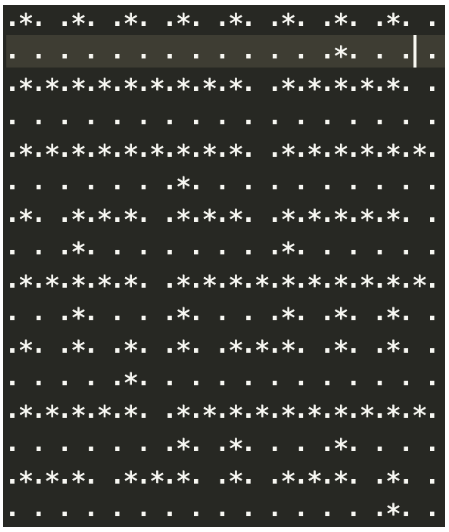
Solution - This is the exact property that a tree has, i.e. unique path between any two vertices. Now, we know that each of the tree search algorithms correspond to a unique search tree (e.g. a BFS search tree). Thus, we randomly construct a search tree using the generation algorithm (MST for Kruskals and BFS Tree with BFS) and make the equivalent half-resolution (fig-2.1) maze for it.
Now, since each cell in Spanning Tree is reachable ⇔ We can construct the half-resolution maze for it with each cell reachable and scale it up to the full resolution with 2-d walls.

Analysing the resolution conversion - For example, consider



```
_._._._. ._|_. |
_._._._. ._._._|
 ._. |_. ._._. |
_|_. ._._|_._._|
 | . | ._| | | |
_._| ._._._._._|
_. ._| | ._| . |
_._._._._._._|_|
```

(a) *8x8 low resolution maze generated*



```
.*. .*. .*. .*. .*. .*. .*. .*. .
. . . . . . . . . . . . .*. . .|.
.*.*.*.*.*.*.*.*.*. .*.*.*.*.*. .
. . . . . . . . . . . . . . . . .
.*.*.*.*.*.*.*.*.*. .*.*.*.*.*.*.
. . . . . . .*. . . . . . . . . .
.*. .*.*.*. .*.*.*. .*.*.*.*.*. .
. . .*. . . . . . .*. . . . . . .
.*.*.*.*.*. .*.*.*.*.*.*.*.*.*.*.
. . .*. . . .*. . . .*. .*. .*. .
.*. .*. .*. .*. .*.*.*. .*. .*. .
. . . . .*. . . . . . . . . . . .
.*.*.*.*.*. .*.*.*.*.*.*.*.*.*.*.
. . . . . . .*. .*. . . .*. . . .
.*.*.*. .*.*.*. .*. .*.*.*. .*. .
. . . . . . . . . . . . . . . .*. .
```

(b) *the corresponding 16x6 maze*

Figure 2.2: *An example showing the correspondence between our high-res and low-res mazes*

2.2 Maze Generation using MPI

2.2.1 Kruskals

1. We divided the rows among the different MPI processes as contiguous blocks of rows.
2. Implemented **Disjoint Set Unions** to implement Kruskal's algorithm to generate MST on each block of rows in parallel. Used `Shuffle` function to randomly sort edge list, so that a random maze is generated each time.
3. Used `MPI_Gather` to send MST formed in each process to process 0. Added a random edge between vertices of adjacent processes to connect the various MSTs and form a MST for the overall graph.

2.2.2 BFS

1. We divided the rows among the different MPI processes as contiguous blocks of rows.
2. Randomly chose a node within the nodes available to the process as a source node, and applied BFS algorithm to the nodes available locally. We thus get a MST for each block of rows.
3. Used `MPI_Gather` to send MST formed in each process to process 0. Added a random edge between vertices of adjacent processes to connect the various MSTs and form a MST for the overall graph.

3 Solving

3.1 DFS

1. In this we use the algorithm in Pacheco Chapter 6 to divide the work amongst different processes with the help of stack.
2. We do so by storing every child on the stack and popping when no valid path is found.
3. We use MPI Calls to communicate this information.
4. At the beginning of every iteration we check whether there exists some free processor with no work if then we share the top element of max filled processor to that processor.
5. This way the work is divided between the processors. And no processor is idle.
6. At the end we then merge the parent arrays to find a valid path to the source and print the updated maze.

3.2 Dijkstra

1. In this, we use the algorithm described in class, dividing the processors such that every vertex i, j is assigned to the processor $(i \times 64 + j) \mod 4$.
2. We then declare local frontiers, and initialize one of them then use allreduce over sz and start the while loop until total size of all frontiers is greater than zero.
3. We then compute the adjacent vertices for every to element in the priority queue. Once that is done, we use mpi_send and mpi_recv to communicate between different processors.
4. Once all communication is done all processors update their priority queues, and size is recomputed.
5. After the while loop all the processors send the distance related to their vertex to the 0 processor.
6. Once that is done the root computes the path and prints the path.
7. We observe that this would be identical to BFS since all the weights are the same.

4 Code and MPI Details

4.1 Maze Generation:

4.1.1 Kruskal:

- **Synchronization:** We used `MPI_Barrier` to wait for all processes to complete, before broadcasting the resulting final MST to all processes for solving.
- **Optimizations:** We do not maintain an adjacency matrix for the graph representation, since it will be $O(V^2)$ in size. We only maintain a local edge list, which is a vector of pair of integers, each element representing the vertices of an edge. It will be $O(V)$ in size.

Time Complexity Analysis:

- **Sequential Time Complexity:**
  If there is only one process, time complexity will be $T_s = O(V)$ where V is the number of vertices, i.e. Maze size x Maze Size.
- **Parallel Time Complexity:**
  - **Computation Time:** The Computation time will be $T_{\text{compute}} = O\left(\frac{V}{p}\right)$
  - **Communication Time:** We use the Gather function once. Time complexity to send one edge = $O(\log p)$, where p is the number of processes. A tree will have $E = |V|$ edges, so total communication time will be $O(V \log p)$.
  - **Total Parallel Time Complexity:** $O\left(\frac{V}{p}\right) + O(V \log p)$
  - **Speedup:**

$$Speedup = \frac{O(V)}{O\left(\frac{V}{p}\right) + O(V \log p)}$$

$$Speedup = \frac{1}{O\left(\frac{1}{p}\right) + O(\log p)}$$

  - **Efficiency:**

$$\text{Efficiency} = \frac{1}{1 + O(p \log p)}$$

4.1.2 BFS:

- **Synchronization:** We used `MPI_Barrier` to wait for all processes to complete, before broadcasting the resulting final MST to all processes for solving.
- **Optimizations:** We do not maintain an adjacency matrix for the graph representation, since it will be $O(V^2)$ in size. We only maintain a local edge list, which is a vector of pair of integers, each element representing the vertices of an edge. It will be $O(V)$ in size.

  Time Complexity Analysis:

- **Sequential Time Complexity:**
  If there is only one process, time complexity will be $T_s = O(V)$ where V is the number of vertices, i.e. Maze size x Maze Size.
- **Parallel Time Complexity:**
  - **Computation Time:** The Computation time will be $T_{\text{compute}} = O\left(\frac{V}{p}\right)$
  - **Communication Time:** We use the Gather function once. Time complexity to send one edge = $O(\log p)$, where p is the number of processes. A tree will have $E = |V|$ edges, so total communication time will be $O(V \log p)$.
  - **Total Parallel Time Complexity:** $O\left(\frac{V}{p}\right) + O(V \log p)$
  - **Speedup:**

$$Speedup = \frac{O(V)}{O\left(\frac{V}{p}\right) + O(V \log p)}$$

$$Speedup = \frac{1}{O\left(\frac{1}{p}\right) + O(\log p)}$$

  - **Efficiency:**

$$\text{Efficiency} = \frac{1}{1 + O(p \log p)}$$

4.2 Maze solving

4.2.1 DFS :

- **Synchronization:** We used texttt MPI_Barrier at various places wherever we required the processes to run in synchronous manner.
- **Optimization:** Just as in generation in maze solving, we never store any sort of adjacency matrix, since it will be $O(V^2)$ in size and always use edges by checking

the maze which is passed by reference. Thereby optimisizing on the use of storage and time both.

Time Complexity Analysis:

- **Sequential Time Complexity:**
  If there is only one processor, time complexity to solve would be be $O(V + E)$ in our case though E = $O(V)$ where V is the number of vertices and E is the number of edges.
- **Parallel Time Complexity:**

  - **Computation Time:** The Computation time will be $T_{\text{compute}} = O\left(\frac{V}{p}\right)$
  - **Communication Time:** We use the AllGather and AllReduce functions to send the sz of stack in every iteration. Time complexity due to them $= O(\log p)$, where p is the number of processes, apart from that at the end we use allgather, send, recv to send the entire parent array for all nodes which cause $O(V \log p)$ time so total communication time will be $O(V \log p)$.
  - **Total Parallel Time Complexity:** $O\left(\frac{V}{p}\right) + O(V \log p)$
  - **Speedup:**

$$Speedup = \frac{O(V)}{O\left(\frac{V}{p}\right) + O(V \log p)}$$

$$Speedup = \frac{1}{O\left(\frac{1}{p}\right) + O(\log p)}$$

  - **Efficiency:**

$$\text{Efficiency} = \frac{1}{1 + O(p \log p)}$$

4.2.2 Dijkstra :

- **Synchronization:** We used texttttMPI_Barrier at various places wherever we required the processes to run in synchronous manner.
- **Optimization:** Just as in generation in maze solving, we never store any sort of adjacency matrix, since it will be $O(V^2)$ in size and always use edges by checking the maze which is passed by reference. Thereby optimisizing on the use of storage and time both.

Time Complexity Analysis:

- **Sequential Time Complexity:**
  If there is only one processor, time complexity to solve would be be $O((V+E)\log V)$ in our case though E = $O(V)$ where V is the number of vertices and E is the number of edges. Therefore Time complexity is $O((V)\log V)$

- **Parallel Time Complexity:**
  - **Computation Time:** The Computation time will be $T_{\text{compute}} = O\left(\frac{V \log V}{p}\right)$ due to the priority queue.
  - **Communication Time:** We use the AllGather and AllReduce functions to perform communication and adjacent vertices. Time complexity due to them $= O(\log p)$ in every iteration, where p is the number of processes, apart from that at the end we use allgather, send, recv to send the entire parent array for all nodes which cause $O(V \log p)$ time so total communication time will be $O(V \log p)$.
  - **Total Parallel Time Complexity:** $O\left(\frac{V \log V}{p}\right) + O(V \log p)$
  - **Speedup:**

$$Speedup = \frac{O(V \log V)}{O\left(\frac{V \log V}{p}\right) + O(V \log p)}$$

$$Speedup = \frac{O(\log V)}{O\left(\frac{\log V}{p}\right) + O(\log p)}$$

  - **Efficiency:**

$$\text{Efficiency} = \frac{O(\log V)}{O(\log V) + O(p \log p)}$$

5

Optimisations and Results

5.1 Optimisations

Apart from the previously mentioned Optimisations in *Code and MPI Details* chapter, we performed the following optimisations -

1. Usage of **Edge Lists** rather than Adjacency Matrices that stores in $O(|E|) = O(|V|)$ rather than $O(|V^2|)$.
2. Not sending the entire maze to each process but only the part of memory needed by each processor
3. Pass by Reference for Maze Solving
4. We seeded the randomisation with a function of (my_rank) to ensure different generation on different processes.

5.2 Results

5.2.1 Observations

1. We notice that Dijkstra for a graph with uni-weighted edges is the exact same as Breadth-First-Search.
2. The theoretical speedup is bounded by 4x for all.
3. For small maze sizes, the MPI parallelised version practically runs slower due to the expensive MPI functions overhead compared to Operations of very small linear order $O(|V|)$. However for bigger maze sizes, parallelized MPI versions end up being better eventually as the overhead becomes insignificant in the long run.

5.2.2 Maze Characteristics

Mazes generated via Breadth-First Search (BFS) and Kruskal's algorithm differ in several properties (and looks) due to the inherent nature of the algorithms and the trees they produce -

- **Path Lengths:**
  - **BFS:** Mazes generated using BFS tend to have shorter solution paths because BFS prioritizes exploring the nearest nodes first, leading to more direct routes between start and end points.
  - **Kruskal's Algorithm:** The generated maze typically has longer solution paths because it focuses on creating a spanning tree that connects all cells without considering the shortest path between specific points.
- **Complexity:**
  - **BFS:** The complexity of generating a maze using BFS is typically higher because it involves exploring all possible paths from the start point to the end point.
  - **Kruskal's Algorithm:** Kruskal's algorithm tends to be less complex because it focuses on connecting cells randomly while ensuring that no cycles are formed, which can result in a simpler maze structure.
- **Topology:**
  - **BFS:** Mazes generated with BFS often have a more regular structure, with corridors and dead-ends arranged in a more systematic pattern.
  - **Kruskal's Algorithm:** The resulting maze from Kruskal's algorithm may have a more irregular topology, with a higher likelihood of open spaces, loops, and long corridors.
- **Symmetry:**
  - **BFS:** Mazes generated with BFS may exhibit more symmetry due to the systematic exploration of adjacent cells.
  - **Kruskal's Algorithm:** The resulting maze can be less symmetrical, with variations in wall placement and corridor lengths depending on the randomly selected edges.