Assignment – 1

3.1 : Brief Approach to Solve the Problem 3.1: Escape the collapsing maze

The goal is to compute the shortest distances from the **exit node (source node)** to all starting nodes in a distributed environment using MPI. Here's the step-by-step breakdown of the approach:

---

1. Reverse Directed Edges

- The directed edges in the input are reversed to simulate the problem as a BFS from the **exit node** (source node) to the explorers (starting nodes).
- This allows us to solve the problem as a **single-source shortest path** problem starting from the exit node.

---

2. Input Parsing and Setup

- The root process (`rank 0`) reads the input graph, starting nodes, exit node, and blocked nodes:
    - **Graph Representation**: The graph is stored as an adjacency list where each vertex has a list of neighbors `(v, d)` where `v` is the neighbor and `d` denotes whether the edge is directed (`0`) or undirected (`1`).
    - **Blocked Nodes**: These nodes are skipped during BFS traversal.
    - **Starting Nodes**: These are the explorers, whose shortest distance from the exit node needs to be computed.
- The input graph, blocked nodes, and starting nodes are broadcasted to all processes.

---

3. Partition the Graph

- The vertices are partitioned across processes:
    - Each process is responsible for a **contiguous block of vertices** (`local_start` to `local_end`) based on its rank.
    - If the number of processes exceeds the number of vertices, some processes will handle zero vertices.

---

4. Parallel BFS Traversal

- **Initialization**:
    - Each process initializes a local distance array (`local_distances`) to `INT_MAX`.
    - The process responsible for the **exit node** sets its distance to `0` and begins BFS from this node.
- **BFS Iteration**:
    - Each process iterates over its local vertices (`local_start` to `local_end`).
    - For each vertex `u`:
        - Traverse its neighbors `v` and propagate the distance if `u + 1 < v`.
        - Skip blocked nodes during traversal.
    - If any process makes updates to distances, the `local_active` flag is set to `true`.
- **Synchronization**:
    - All processes use `MPI_Allreduce` to combine their local distance arrays into a global distance array (`MPI_MIN` for the shortest path).
    - Use `MPI_Allreduce` to synchronize the `local_active` flags to determine if further BFS iterations are needed.
- **Termination**:
    - BFS continues until no process is active (`global_active == false`).

---

5. Handle Edge Cases

- **Blocked Exit Node**:
    - If the exit node is blocked, all distances remain `INT_MAX`, and the result for all starting nodes is `-1`.
- **Disconnected Graph**:
    - If any starting node is not reachable from the exit node, its distance remains `INT_MAX`, and the result for that node is `-1`.

---

6. Result Gathering

- After BFS completes, the root process (`rank 0`):
    - Maps the computed distances for all starting nodes.
    - Converts `INT_MAX` to `-1` for nodes that are unreachable.

Time Complexity:

- **BFS Traversal**:
    - Each edge and vertex is processed once, leading to a time complexity of O(V+E).

- **MPI Communication**:
  - Each `MPI_Allreduce` operation is O(log P), where P is the number of processes.

Space Complexity:

- Each process stores:
  - **Graph**: O(V+E)
  - **Distance Array**: O(V)

Message Complexity of the Solution

The **message complexity** refers to the total number of messages exchanged between MPI processes during the execution of the algorithm. For the given parallel BFS solution, the message complexity arises primarily from:

1. **Broadcast Operations**:
   - Broadcasting the graph structure, starting nodes, and blocked nodes.
   - Broadcasting the global distance array in each BFS iteration.
2. **Reduction Operations**:
   - Performing `MPI_Allreduce` to:
     - Aggregate the minimum distances across all processes (`MPI_MIN` for distances).
     - Synchronize the `local_active` flag to determine if BFS should continue (`MPI_LOR` for the active state).

---

1. Broadcast Message Complexity

- **Graph Broadcast**:
  - The flattened graph structure is broadcast once from the root process (`rank 0`) to all other processes.
  - Message Complexity: O(V+E), as the graph is broadcast as a flat array.
- **Blocked Nodes Broadcast**:
  - The blocked nodes are broadcast once as a list.
  - Message Complexity: O(B), where B is the number of blocked nodes.
- **Starting Nodes Broadcast**:
  - The starting nodes are broadcast once.
  - Message Complexity: O(K) where K is the number of starting nodes.

Total Broadcast Complexity: O(V+E+B+K)

2. Reduction Message Complexity

For each BFS iteration:

- **Distance Reduction**:
  - `MPI_Allreduce` combines the local distances into the global distance array using the `MPI_MIN` operation.
  - Each process sends its local distance array of size V, and the complexity of this operation is: $O(V \cdot \log(P))$ where P is the number of processes.
- **Active State Reduction**:
  - `MPI_Allreduce` checks the `local_active` flag across all processes using the `MPI_LOR` operation.
  - The complexity of this operation is: $O(\log(P))$

If the BFS completes in LLL levels (iterations of BFS), where LLL is the diameter of the graph, then:

- Distance Reduction Complexity: $O(L \cdot V \cdot \log(P))$
- Active State Reduction Complexity: $O(L \cdot \log(P))$

Total Reduction Complexity: $O(L \cdot V \cdot \log(P)) + O(L \cdot \log(P))$

3. Overall Message Complexity

Combining the broadcast and reduction complexities, the total message complexity of the solution is:

$O(V+E+B+K)+O(L \cdot V \cdot \log(P))+O(L \cdot \log(P))$

Where:

- V: Number of vertices.
- E: Number of edges.
- B: Number of blocked nodes.
- K: Number of starting nodes.
- L: Diameter of the graph (number of BFS levels).
- P: Number of processes.

Dominant Term

For large-scale graphs, the dominant term is the reduction of the distance array: O(L·V·log(P))

Thus, the **overall message complexity** can be approximated as: O(L·V·log(P))

This reflects the communication overhead for synchronizing distances during each BFS iteration in a distributed system.

First for small input : V = 48 vertices and E =148 edges
N = 1 process : t = 0.001266
N = 2 process : t = 0.002483
N = 4 process : t = 0.002215
N = 8 process : t = 0.001672
N = 12 process : t = 0.016091

Second for large input : V = 10000 vertices and E = 9999 edges
N =1 process : t = 10.534600
N = 2 process : t = 9.598703
N = 4 process : t = 9.330239
N = 8 process : t = 8.840641
N = 10 process : t = 10.307092

Brief Approach to Solve the Problem 3.2: Bob and Bouncing Balls

This program is an MPI-based solution for handling particle collisions in a distributed circular grid environment. The following steps explain the approach used in the program:

---

1. Initialization

- The program initializes MPI and determines the rank and size of the processes.
- The grid dimensions (`m` and `n`), number of particles (`k`), number of time steps (`t`), and process boundaries (`portion_length`) are distributed among the processes.

---

2. Particle Distribution

- The root process (`rank == 0`) assigns particles to processes based on their `x`-coordinate and sends the serialized particle data to other processes.

---

3. Particle Movement

- Particles move according to their direction (`L`, `R`, `U`, `D`) using the `move_ahead` function, which ensures circular grid wrapping.
- Particles that cross the boundaries of a process are sent to the next or previous process using MPI communication.

---

4. Collision Handling

- Particles that collide at the same grid location are handled based on the number of particles at that location:
  - **Two-particle collision**: Particles turn 90 degrees to the right using the `collideTwo` function.
  - **Four-particle collision**: Particles reverse their direction using the `collideFour` function.
  - **Three-particle collision**: No direction changes are made.

---

5. Boundary Communication

- Each process sends particles crossing its boundaries to neighboring processes (using MPI `Send` and `Recv`).

- Particles are received by the appropriate neighboring processes and integrated into their local particle map.

---

6. Synchronization

- All processes are synchronized using `MPI_Barrier` before handling collisions to ensure consistency.

---

7. Gathering Results

- After all time steps, each process sends its particle data back to the root process.
- The root process collects, sorts, and outputs the final particle positions and directions.

Highlights of the Logic:

1. **Circular Grid Wrapping**:
    - Ensures particles wrap around the grid using modulo operations.
2. **Efficient Communication**:
    - Only particles crossing process boundaries are communicated, minimizing overhead.
3. **Scalable Design**:
    - The grid is divided dynamically among processes, allowing the solution to scale with the number of processes.

---

1. Time Complexity

Particle Movement (`move_particles_ahead`)

- **Per Particle**:
    - Each particle updates its position using constant time operations (`O(1)`).
    - Checking if the particle crosses a process boundary is also `O(1)`.
- **Total**:
    - For `K` particles, the movement operation is `O(K)`.

Collision Handling (`handle_collisions`)

- **Per Particle**:
    - Particles are grouped by position into a `map` (or hash table), which takes `O(1)` on average per particle.

    - Collision handling for each position takes `O(1)` per group because the group size is constant (at most 4 particles per cell).
- **Total**:
    - Iterating through all particles and handling collisions takes `O(K)`.

MPI Communication

- **Send/Receive Particles**:
    - Each process sends and receives particles crossing boundaries, which is proportional to the number of particles crossing the boundaries, `C`. For `P` processes, each process communicates at most `O(C)` particles to its neighbors.
    - Over all processes, communication takes `O(C * P)`, where `C` is the average number of crossing particles per process.
- **Broadcasting Grid Information**:
    - Grid parameters (`m`, `n`, `k`, `t`, `portion_length`) are broadcast once, taking `O(P)`.

Sorting Final Output

- **On Root Process**:
    - The root process gathers and sorts the particles based on their position and ID. Sorting takes `O(K log K)`.

Total Time Complexity:

- Movement + Collision Handling: `O(K * T)`
- MPI Communication: `O(C * P * T)`
- Final Sorting: `O(K log K)`

Overall:
$O(K*T+C*P*T+K\log K)$

---

2. Space Complexity

Per Process

- Each process stores:
    1. Particles assigned to its grid portion: `O(K / P)` for `K` particles and `P` processes.
    2. Boundary crossing lists (`list_crossed_upperlimit` and `list_crossed_lowerlimit`): At most `O(C)` particles per process.
    3. Position map for collision handling: `O(K / P)` for local particles.
    - **Per Process**: `O(K / P + C)`.

Total Across All Processes

- The total memory usage across all processes is: O(K+C×P)O(K + C \times P)O(K+C×P)

Root Process

- The root process gathers and stores all particles for sorting and final output:
  - **Root Memory**: `O(K)`.

---

3. Message Complexity

Message complexity measures the number of messages sent between processes.

Per Time Step

- **Boundary Communication**:
  - Each process sends and receives the count of particles crossing boundaries (`O(1)` message).
  - Each crossing particle requires a `O(1)` message to be sent to its neighboring process.
  - Total Messages per Time Step: `O(C * P)`.

Broadcasting

- **Grid Information**:
  - Parameters (`m`, `n`, `k`, `t`, portion lengths) are broadcast to all processes once. This is `O(P)` messages.

Gathering Results

- The root process gathers results from all `P` processes at the end, requiring `O(K)` messages in total (each process sends its particle data).

Total Message Complexity:

- **Per Time Step**: `O(C * P)`
- **Total for All Steps**: `O(T * C * P)`
- **Final Gather**: `O(K)`

Small input where M = 5 rows, N = 6 columns , K = 10 balls , T =10 seconds

For N=1 process : t = 0.000501

For N=2 process : t = 0.000688

For N =4 process : t = 0.001046

For N = 8 process : t = 0.001407

For N = 12 process : t = 0.019332

Very Large input where M = 624 rows, N = 321 columns , K = 9093 balls , T =7908 seconds

For N =1 process : t = 211.363679

For N = 2 process : t = 110.205902

For N = 4 process : t = 65.530279

For N = 7 process : t = 45.763963

For N = 11 process : t = 49.180860

For N =12 process : t = 72.827712

Brief Approach to Solve the Problem 3.3: Distributed File System

1. Overview

The solution implements a simplified distributed file system using **MPI** (Message Passing Interface). There are two main roles:

1. **Metadata Server (Rank 0)**
   - Maintains global knowledge (metadata) of all stored files, how each file is divided into chunks, and where each chunk is replicated.
   - Tracks which storage nodes are "alive" based on heartbeats.
   - Processes user commands (upload, retrieve, search, list_file, failover, recover).
2. **Storage Nodes (Ranks 1, 2, 3, …)**
   - Physically store the file chunks they receive from the metadata server.
   - Send periodic heartbeats to inform the metadata server they are still active.
   - Respond to requests from the metadata server for data retrieval or searching.

---

2. Core Components

1. **Chunking and Replication**
   - Files are split into fixed-size chunks (32 bytes, by default `CHUNK_SIZE = 32`).
   - The metadata server replicates each chunk to a number of different storage nodes (default `REPLICATION_FACTOR = 3`).
   - This ensures data redundancy: if one node fails, the file can still be reconstructed from other replicas.
2. **Metadata Management**
   - The server maintains a map of all files, each entry containing:
     - The file name (logical ID)
     - A list of chunks (each chunk has the actual data plus a list of node ranks that store it).
   - When a user uploads a file, the server reads the file locally, splits it, picks target nodes for each chunk, and updates the metadata structure accordingly.
3. **Storage Nodes**
   - Each storage node has an internal map from `(filename, chunkId)` to the chunk data.

- When the server instructs a node to store a chunk, the node simply places it in this local map.
- When the server requests retrieval or search, the node looks up the chunk in this map and performs the requested operation.

4. **Heartbeat Mechanism and Failover**
   - Each storage node runs a **heartbeat thread**: it sends a simple MPI message (containing its own rank) to rank 0 on a fixed interval.
   - The metadata server has a **monitoring thread** that checks when it last received a heartbeat from each node. If too long has passed (exceeding `FAILOVER_THRESHOLD`), the node is considered "dead."
   - **failover** command simulates a node crash, turning off heartbeats from that node and removing it from the active set.
   - **recover** command simulates restarting the crashed node and brings it back into the active set.

---

3. Main Operations

1. **Upload** (`upload <logical_filename> <filepath>`)
   - The metadata server reads the file from disk, breaks it into chunks, and randomly assigns each chunk to `REPLICATION_FACTOR` distinct storage nodes.
   - Each node receiving a chunk stores it locally.
   - The server updates its metadata with the chunk locations and prints a success message along with chunk assignment details.
2. **Retrieve** (`retrieve <logical_filename>`)
   - The server iterates through every chunk for the requested file.
   - It finds the first node that currently appears active for that chunk and asks for its data.
   - After collecting all chunks in order, it reconstructs the original file content and prints it.
3. **Search** (`search <logical_filename> <word>`)
   - The server looks up each chunk for the file and tries to contact one active node holding that chunk.
   - The node searches inside its local chunk data for all occurrences of the search word, offset by the chunk's starting position.
   - The server merges results across all chunks, prints the count of matches, and lists the global offsets.
4. **List File** (`list_file <logical_filename>`)
   - The server prints each chunk ID, how many **active** replicas exist, and which ranks store them.
5. **Failover** (`failover <rank_to_crash>`)
   - Simulates a crash.
   - The crashed node stops sending heartbeats; the server removes it from the active set.
6. **Recover** (`recover <rank_to_restart>`)
   - Simulates restarting the node.
   - The node resumes sending heartbeats, and the server adds it back to the active set.
7. **exit**
   - Terminates the command loop and shuts down.

4. Communication Flow

- **Metadata Server -> Storage Nodes**
    - Used when assigning chunks for upload, requesting chunk data for retrieval, or instructing a node to search locally.
- **Storage Nodes -> Metadata Server**
    - Used for heartbeats, sending back search results, and sending chunk data during retrieval.
- **Internal Concurrency**
    - Metadata server has one thread monitoring heartbeats, and the main thread processes user commands.
    - Each storage node has one heartbeat thread plus one main thread handling incoming requests.

5. Failover Handling

- The system detects failures by monitoring the time since the last heartbeat from each node. If it exceeds a threshold, that node is removed from `activeNodes`.
- Future requests for chunks on that node are skipped in favor of replicas on other nodes.
- The system does not automatically re-replicate chunks onto healthy nodes (this is a possible future improvement).
- A manual **recover** command simulates a node coming back online.

6. Key Takeaways

1. **MPI for Distributed Control**: Rank 0 acts as the coordinator (metadata server), and ranks 1+ act as storage.
2. **Threaded Heartbeats**: Demonstrates how to track node liveness in an asynchronous manner.
3. **Chunk-Based Replication**: Splitting files into 32-byte chunks and replicating them.
4. **Search & Retrieval**: Locating data across multiple chunks/nodes, demonstrating how to aggregate data from distributed sources.
5. **Manual Failover/Recovery**: Illustrates how to test fault scenarios in a distributed system.

Overall, this application showcases a simplified but functional approach for distributed data storage, combining chunk-based replication, node liveness detection, and basic file operations (upload, retrieve, search).