MPI Assignment

Deadline: 11:59pm, 24th Jan

1 Introduction

In this assignment, you will construct distributed solutions to certain problems and implement them using the Message Passing Interface: MPI in any programming language of your choice (C/C++ is strongly recommended owing to the amount of documentation available online).

- Documentation: https://rookiehpc.github.io/mpi/docs/index.html
- Tutorial: `https://rookiehpc.github.io/mpi/exercises/index.html`

2 Instructions

- Answer all the questions.
- You may only use MPI library routines for communicating and coordinating between the MPI processes. You cannot use any form of shared-memory parallelism. The idea is to develop a program that could be deployed on a large message-passing system.
- Any language can be used for all of the following questions. **We recommend C/C++.**
- Do not use any platform specific libraries.
- The code you submit will be run alongside a Profiler to check if you are parallelizing the code and not submitting a sequential solution.
- Input must be read by **only one of your spawned processes** from a file. You may then choose to transfer this data between nodes as you wish. The absolute path of this file will be provided as a command line argument to your executable. The output must be printed to standard output.
- You should test your program with varying number of processes and a sequential program across various input sizes and tabulate the run time reported along with any other observations.
- Your programs should execute with any number of processes between 1 and 12. For those who are not able to run with higher number of processes, use the following command (for C++):
  `mpiexec -np 12 --use-hwthread-cpus --oversubscribe ./a.out`
- Prepare a report containing the solution approach, highlights of your program, and the results obtained and submit the report along with a README file, the source files, and datasets. Bundle all of them as a single zip/tar ball and submit via moodle.
- Any code that is found to be copied from any source will attract penalty.
- Manual evaluations will also be conducted for the assignment, and 5 marks from every question are reserved for viva component(i.e. in total 15 marks from the assignment).

3 Problems

3.1 Escape the collapsing maze (20 points)

Your team is trapped inside a **collapsing ancient maze**! The maze is a vast network of interconnected chambers and corridors, represented as a graph $G = (V, E)$ where:

- Each node $v \in V$ is a chamber.
- Each edge $e \in E$ is a corridor connecting two chambers.

The **exit** of the maze is located at node R. You must guide everyone in your team from their starting positions $S = \{s_1, s_2, \ldots, s_k\}$ to the exit before the maze collapses.

The maze is enormous, and solving the problem quickly is critical. To escape, you must find the **shortest path** from each starting position s_i to the exit R. However, the maze has a few quirks:

- **One-way passages:** Certain corridors can only be traversed in a specific direction.
- **Disconnected zones:** Some chambers cannot be reached at all.

Write a program using **MPI** to help your team to escape the collapsing ancient maze.

3.1.1 Input Format:

Take the input directly from **stdin**.

- *V E (Number of nodes and edges respectively)*
- The next E lines will have the information about the edges with each line having 3 numbers: u v d where u and v are the end-nodes of the edge and d tells about the directionality ($d = 1$ if bidirectional, else 0). Unidirectionality implies you can go from u to v only.
- The next line will have k: the number of explorers.
- The next line will be an array of size k denoting starting nodes of each explorer.
- The next line will have the exit node R.
- The next line will have L: the number of blocked chambers.
- Finally, the last line will be an array of size L denoting blocked chamber indices.

All indices are 0-indexed.

3.1.2 Output Format:

Write the output directly to **stdout**.

An array of size k where each element represents the shortest distance to R from the corresponding starting chamber. If no path exists from that point, output -1 in that place.

3.1.3 Constraints:

$1 \leq V, E \leq 10^5$

$0 \leq u, v, k, L, R \leq V - 1$

$d \in \{0, 1\}$

Note: The starting nodes of explorers will not be blocked.

3.1.4 Example

Input

```
5 6
0 1 1
0 2 1
1 2 1
1 3 1
2 4 1
3 4 0
4
1 4 2 0
4
1
3
```

Output

```
2 0 1 2
```

Explanation:

- The first explorer can go to node 2 and then to 4.
- The second explorer is already at the exit node.
- The third explorer can directly go to 4.
- The fourth explorer will go to node 2 and then to 4.

3.2 Bob and Bouncing Balls (35 points)

In a vibrant world filled with bouncing balls, Bob finds himself in the midst of an exhilarating game that tests his strategic thinking and quick reflexes. The game unfolds on a dynamic 2D grid, where each ball is not just a simple object but a character with its own personality and direction. As the timer ticks down over a series of time steps, Bob must navigate the complexities of ball movements while adhering to the intricate rules of collision and direction changes.

The 2D grid has N rows numbered from 0 to N-1 from top to bottom and M columns numbered from 0 to M-1 from left to right. Bob's ultimate goal is to manage multiple balls simultaneously, each moving in its designated direction—up, down, left, or right. In particular, there are K different balls at various positions in the grid each having a particular direction. The thrill escalates as he realizes that these balls can move and collide in unpredictable ways given by the following rules.

- During each time step, every ball moves by one unit in its respective direction and changes direction if necessary. Every ball maintains its velocity after moving; however, it cannot occupy the same grid point as another ball moving in the same direction.
- If exactly two balls collide at a point from any directions, they always bounce off at 90 degrees (towards right hand side). Suppose ball 1 at point (x-1, y) going D and ball 2 at point (x+1, y) going U at time (t-1). At time t, both collide at (x, y) and the directions will become L and R respectively. At time (t+1), ball 1 would be at point (x, y-1) going L and ball 2 at point (x, y+1) going R.
- If exactly four balls collide at a point from all the directions, they always bounce off at 360 degrees (move in opposite direction).
- Suppose ball 1 at point (x-1, y) going D, ball 2 at point (x, y+1) going L, ball 3 at point (x+1, y) going U and ball 4 at point (x, y-1) going R at time (t-1). At time, all the four balls would collide at (x, y) and the directions will become U, R, D and L respectively. At time (t+1), ball 1 would be at point (x-1, y) going U, ball 2 at point (x, y+1) going R, ball 3 at point (x+1, y) going D and ball 4 at point (x, y-1) going L.
- Now, you might be wondering what might happen if exactly three balls collide at a point from any directions. Unfortunately, in such a case, they continue in the same direction without any deflection.
- The grid is circular. If any ball reaches the boundary of the grid, it reaches the opposite end in the next time step. Suppose a ball at point (x, 1) going L at time (t-1). It would be at point (x, 0) going L at time t. At time (t+1), it would be at point (x, m-1) going L.

Bob understands that the above game runs for T time steps. As Bob finds it difficult to understand the rules of the game, he wants your help to play the game. Given the initial configuration of the grid, your task is to write a program using **MPI** that simulates the game and output the final configuration of the balls on the grid.

3.2.1 Input Format

Take the input directly from **stdin**.

The first line contains four integers: N (number of rows), M (number of columns), K (number of balls), and T (number of time steps).

Each of the next K lines contains three values: integer x (initial x-coordinate), integer y (initial y-coordinate) and character c (initial direction).

Note: No two balls start at the same position with the same initial direction.

3.2.2 Output Format

Write the output directly to **stdout**.

Output K lines, the i^{th} line containing the final position (x-coordinate, y-coordinate) and direction of the i^{th} ball after T time steps.

Note: The relative order of the balls in the output should be same as the input.

3.2.3 Constraints:

$1 \leq N * M \leq 2 \cdot 10^6$

$1 \leq K * T \leq 2 \cdot 10^6$

$0 \leq x < N$

$0 \leq y < M$

$c \in \{\text{'U'}, \text{'D'}, \text{'L'}, \text{'R'}\}$

3.2.4 Example

Input 1

```
3 4 3 2
0 0 R
2 3 L
1 1 U
```

Output 1

```
1 1 D
2 1 L
0 2 R
```

Input 2

```
8 8 10 10
1 2 D
5 4 U
3 0 R
2 5 L
3 4 L
7 0 L
2 5 U
4 4 D
5 2 U
7 0 R
```

Output 2

```
5 6 D
3 4 U
2 1 D
7 6 D
7 6 L
0 5 U
2 1 L
```

```
6 4 D
4 1 U
7 2 R
```

3.3 Avengers Distributed File System Quest (45 Points)

In a mission to safeguard Earth's most critical data, the Avengers must securely store and manage their vast amount of information. This requires a robust Distributed File System (DFS), the Avengers Distributed File System (ADFS), that ensures redundancy, reliability, and resilience. Your task is to help Iron Man, who serves as the central metadata server, manage this distributed system with the help of various other Avengers, who serve as storage nodes in the system.

Note: Metadata server would only store the metadata related to files along with other indexes for locating chunks. It would not be storing any data related to files on it's own. The onus of storing chunks is completely on the storage servers.

You are tasked with implementing a distributed file system using **MPI** where:

- **Iron Man (Rank 0)** acts as the metadata server.
- **Avengers (Ranks 1 to N-1)** serve as storage nodes.
- The system handles file uploads, downloads, and searches, with built-in load balancing and replication.
- You are also required to implement failover and recovery options to simulate storage node failures and recover the storage nodes respectively. (This will be explained later).

Note: Here, **N** represents the number of processes which will be spawned during execution of the program. For this problem, assume that a minimum of 4 processes will be spawned initially.

Your system should be functional even in case of few failures. You can assume that the metadata node will always be up (0 downtime).

3.3.1 Features to Implement (40 points, 5 for viva)

1. **File Partitioning and Three-Way Replication (5 points)**
   - Files are split into fixed-size chunks (32 bytes each).
   - Each chunk is replicated on three distinct nodes to ensure that the system can continue to function even during storage node failures.
   - Note that there is no requirement to maintain the replication factor in case of failures.
2. **Load Balancing (5 points)**
   - Implement basic load distribution to evenly distribute chunks across nodes.
   - Ensure that no node holds more than one replica of the same chunk.
3. **Heartbeat Mechanism** (**Points combined with Feature 4**, Refer this)
   - Each node sends a periodic heartbeat to the metadata server.
   - If a node fails to send a heartbeat, it is marked as down.
   - Keep the failover interval to be 3 seconds, i.e., if the metadata server does not receive a heartbeat from a node for more than 3 seconds, it will mark that node as down.
   - **Hint:** This can be done by using a separate thread for handling heartbeats.
4. **Failover and Recovery Operations (15 points, 7.5 + 7.5)**
   - **Failover**: Stops requests and heartbeat signals for a specific node rank.
   - **Recover**: Restores the ability of a node to serve requests and resume heartbeat signals.
5. **File Operations (15 points, 4 + 4 + 7)**
   - **Upload**: File is partitioned and chunks are stored with replication.

- **Download**: Chunks are retrieved from nodes and the file is reassembled.
- **Search (Distributed)**: Locate nodes holding specific file chunks.

6. **Error Handling**
   - In case of any failure during operations, output `-1` to indicate an error.
   - For example, if a request comes for searching or downloading a non-existent file, the output should be -1. Ensure all edge cases are handled.

3.3.2 Input Format

Take the input directly from **stdin**. The input format for each command is as follows:

- `upload file_name absolute_file_path`: Upload a file to the distributed file system. The file is partitioned into chunks and replicated across storage nodes. Note that more than one chunks for a file can be present at a particular node.
- `retrieve file_name`: Retrieve a file from the distributed file system. The system retrieves the necessary chunks from the storage nodes and reassembles the file. This functionality should work unless there is a chunk which does not have any replica which is alive.
- `search file_name word`: Search for a specific word in a file. The system locates the offsets containing the word and returns the results. This functionality should work unless there is a chunk which does not have any replica which is alive.
- `list_file file_name`: List all nodes storing chunks of a file. The output includes the number of nodes holding the chunks and their respective ranks, ordered by chunk numbers. Each chunk can be stored on up to three nodes.
- `failover rank`: Simulate the failure of a storage server node by specifying its rank. This stops the node from sending heartbeats and processing requests. This method should also lead to dropping of that rank of process from all existing chunks i.e. when we now run `list_file file_name`, it should no longer have the rank of this process in the output.
- `recover rank`: Recover a previously failed storage server node by specifying its rank. This restores the node's ability to send heartbeats and process requests. This would also lead to addition of the rank of process back to the chunk locations in the metadata server (See Test Case 2 for a better understanding).
- `exit`: Terminate the program. Each test case will end with this command so that the program can gracefully terminate.

3.3.3 Output Format

Write the output directly to **stdout**. The output format for each command is as follows:

- `upload file_name absolute_file_path`: Output 1 along with the output of list_file i.e. mapping from chunks to their respective locations if the operation is successful (see Sample 1). If an error occurs, output `-1`.
- `retrieve file_name`: The system outputs the reassembled file's content. If the file does not exist or an error occurs, output `-1`. If there exists atleast one chunk which does not have any replicas available, then also output -1, i.e. the command should not go through.
- `search file_name word`: The system outputs the occurrences of the word in the file, including the offsets where it was found. If an error occurs, output `-1`. Otherwise, output number of offset in the first line followed by the offsets in the next line. In case number of instances are 0, print a new line before continuing as well. If there exists atleast one chunk which does not have any replicas available, then also output -1, i.e. the command should not go through.

- `list_file file_name`: For each chunk from 0 to M-1 (0 based indexing), the program outputs the number of nodes holding the chunks and their ranks in the following format: `chunk_number num_nodes node_ranks`. If the file does not exist or an error occurs, output `-1`.
- `failover rank`: If an error occurs, output `-1`, otherwise output `1`.
- `recover rank`: If an error occurs, output `-1`, otherwise output `1`.
- `exit`: No output is expected.

For any additional error cases that exist, you have to output -1 as well (For example, trying to upload a file with the same name as one which already exists in the system).

3.3.4 Additional Notes

File Name would be having ".txt" extension.

Relative File Path (w.r.t the program 3.cpp, see the Submission Format.) would be provided wherever applicable. This means that if the file is present at the path ¡RollNumber¿/scripts/3/testcases/a.txt then the path provided would be ../scripts/3/testcases/a.txt i.e. relative to the 3.cpp file.

A valid rank would be provided wherever applicable.

In `search` functionality, only offsets for exact word matches are to be reported.

Send heartbeat messages every second or two (or even less than that is fine), however the check interval for marking a node as down for the metadata server would be kept as 3 seconds.

The size of a word in the testcases would not exceed 32 bytes.

No 2 files which will be uploaded will have the same name.

You can assume that the maximum size of each file would be of the order of 1-2 mb.

Note that the marks distribution given is just a rough idea of the different weightages of the test cases. There would be large combined test cases as well, due to which there might be slight variations.

3.3.5 Example

For both testcases, assume that 5 processes are running (1 for metadata server and 4 for storage server)

a.txt

```
abcd efgh ijkl abcd abcd defg dsds abcddufhef ebhfeihf udefheifh abcde
```

Input 1

```
upload a.txt testcases/a.txt
list_file a.txt
failover 1
failover 2
retrieve a.txt
search a.txt abcd
exit
```

Output 1

```
1
0 3 1 2 3
```

```
1 3 1 2 4
2 3 1 3 4
0 3 1 2 3
1 3 1 2 4
2 3 1 3 4
1
1
abcd efgh ijkl abcd abcd defg dsds abcddufhef ebhfeihf udefheifh abcde
3
0 15 20
```

Input 2

```
upload a.txt testcases/a.txt
failover 1
list_file a.txt
recover 1
list_file a.txt
exit
```

Output 2

```
1
0 3 1 2 3
1 3 1 2 4
2 3 1 3 4
1
0 2 2 3
1 2 2 4
2 2 3 4
1
0 3 1 2 3
1 3 1 2 4
2 3 1 3 4
```

Explanation (Test Case 2):

- Note that the given input in a.txt is of size 70 bytes and would hence be chunked into 3 parts.
- The failover of node with rank 1 causes it's removal from the metadata server.
- The recovery of this node restores it in the stored metadata.

4 Submission Format:

We'll be automating the checking, so make sure you deal with all the edge cases and comply with all the Input-Output specifications and the directory structure given in the submission guidelines below. Not following the given requirements would result in a heavy penalty.

We would be releasing checker scripts in a few days, using which you can easily test your submission format along side the sample inputs provided. Any submission format errors would thus attract huge penalties as we would be providing the checker scripts beforehand.

Any modifications in the assignment or any updates will be given via **Moodle**.

Your submission is expected to be a **<RollNumber>.tar.gz** file containing a directory with the same name as your roll number that holds the following files:

- A directory for each of the mentioned problems with the name: **<ProblemNumber>** containing one source file named **<ProblemNumber>** (e.g. 1.cpp, 1.py)
- You are also required to submit a report, named **Report.pdf** in the root directory of your project, with the following data for every problem you attempt:
  - The total time complexity of your approach
  - The total message complexity of your approach
  - The space requirements of your solution
  - The performance scaling as you went from 1 to 12 processes (use a large enough test case for this, applicable for 3.1 and 3.2)
  - For 3.3, mention your approach in the report and for testing use atleast 4 processes.

Example structure(after file is unzipped):

```
2021101001
├── 1
│   └── 1.cpp
├── 2
│   └── 2.cpp
├── 3
│   └── 3.cpp
└── Report.pdf
```