OPERATING SYSTEMS PROJECT

Umema Ashar 22I-2036
Emad Malik 22I-2072
Muhammad Haider 22I-1913

Contents

MapReduce Framework: Project Report

Introduction:

This project demonstrates the implementation of a **MapReduce Framework** in **C++**, simulating distributed data processing on a single machine. The framework processes input data in parallel, aggregates intermediate results, and provides the final output using **operating system concepts** such as multithreading, synchronization, and inter-process communication through **named pipes**.

The MapReduce process includes three major phases:

1. **Map Phase**: Processes the input data into intermediate key-value pairs.
2. **Shuffle Phase**: Groups key-value pairs by their keys.
3. **Reduce Phase**: Aggregates the grouped pairs to produce the final result.

Code Analysis:

Mapper Program:

The mapper.cpp performs the following tasks:

1. **Splitting**:
    - The input text is split into words using istringstream and is divided into smaller chunks for parallel processing by threads.
    - Each thread processes a portion of the input.
2. **Mapping**:
    - Each thread generates intermediate key-value pairs of the format (word, 1). Here, word represents a unique word, and 1 indicates its initial count.
3. **Shuffling and Writing to Pipe**:
    - After mapping, the generated key-value pairs are sent to the reducer using a **named pipe** (MeraPyaraMapReducePipe).
    - A **mutex (safeLock)** is used to synchronize access to the pipe, ensuring safe concurrent writes by multiple threads.

Reducer Program:

The reducer.cpp performs the following tasks:

1. **Reading from Pipe**:
    - The reducer reads key-value pairs from the named pipe using read and parses the data into individual words and counts.
2. **Reducing**:
    - Key-value pairs are aggregated using a **map<string, int>**.
    - For each word, its counts are summed to produce the final total count.
3. **Final Output**:
    - The aggregated key-value pairs (word and total count) are printed as the final output.

Execution Flow

Below is the high-level flow of the MapReduce process implemented in C++:

1. **Input Splitting**:

   The input text is divided into smaller chunks, each processed by a separate thread.

2. **Mapping**:

   Each thread processes its chunk to generate intermediate key-value pairs.

3. **Shuffling**:

   The generated pairs are grouped by key and sent to the reducer through a named pipe.

4. **Reducing**:

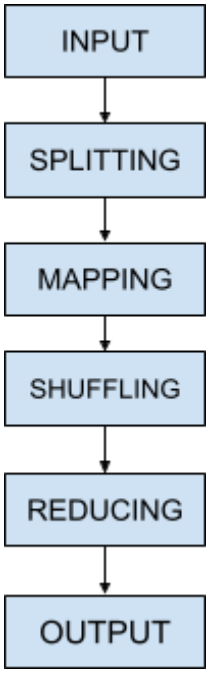
   The reducer aggregates the counts for each unique key.

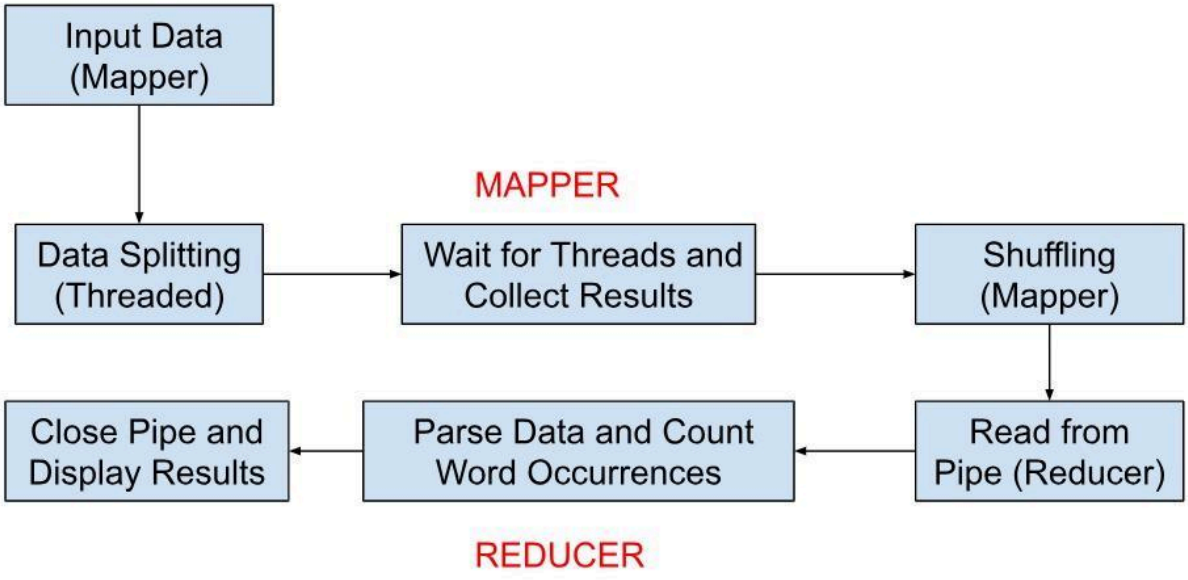
5. **Final Output**:

   The final word count for each unique word is displayed.

Flow Diagram

Here is the flow diagram representing the MapReduce process implemented in this project:





Test Cases:

Below are some test cases to validate the MapReduce framework:

Input	Intermediate Output (Mapper)	Final Output (Reducer)
Deer Bear River	Deer 1, Bear 1, River 1	Deer: 1, Bear: 1, River: 1
Car Car River	Car 1, Car 1, River 1	Car: 2, River: 1
Deer Car Bear Deer	Deer 1, Car 1, Bear 1, Deer 1	Deer: 2, Car: 1, Bear: 1
Bear Bear River River	Bear 1, Bear 1, River 1, River 1	Bear: 2, River: 2
Deer Bear River Car Car River	Deer 1, Bear 1, River 1, Car 1, Car 1, River 1	Deer: 1, Bear: 1, River: 2, Car: 2

Example Execution

1. **Input**:

   Deer Bear River Car Car River Deer Car Bear

2. **Mapper Output (Intermediate)**:

   Deer 1 Bear 1 River 1 Car 1

3. **Reducer Output (Final)**:

   Bear: 2 Car: 3 Deer: 2 River: 2

```
Enter the Input Text:  Deer Bear River Car Car River Deer Car Bear

Data Splitting Successful.
Data Mapping Successful.
Data Shuffling Successful.
```

```
Data Received From Mapper: Bear 1
Data Received From Mapper: Bear 1
Data Received From Mapper: Car 1
Data Received From Mapper: Car 1
Data Received From Mapper: Car 1
Data Received From Mapper: Deer 1
Data Received From Mapper: Deer 1
Data Received From Mapper: River 1
Data Received From Mapper: River 1
```

```
Data Sent To Reducer: Bear 1
Data Sent To Reducer: Bear 1
Data Sent To Reducer: Car 1
Data Sent To Reducer: Car 1
Data Sent To Reducer: Car 1
Data Sent To Reducer: Deer 1
Data Sent To Reducer: Deer 1
Data Sent To Reducer: River 1
Data Sent To Reducer: River 1

Data Sent to Reducer Successfully.
```

```
Data Received From Mapper Successfully.
Data Reducing Successfully.

------ Final Result ------
Bear 2
Car 3
Deer 2
River 2
```

Implementation Approach

1. **Parallelism**:
    - **Threads** are used in the mapper to process input in parallel. Each thread processes a portion of the input, creating key-value pairs.
2. **Synchronization**:
    - A **mutex (safeLock)** ensures thread-safe access to the named pipe, allowing multiple threads to write concurrently without conflicts.
3. **Inter-Process Communication**:
    - The **named pipe (MeraPyaraMapReducePipe)** facilitates communication between the mapper and reducer processes.
4. **Data Aggregation**:
    - The reducer uses a **map<string, int>** to store and aggregate word counts efficiently.
5. **Error Handling**:
    - The implementation includes error handling for pipe operations (e.g., opening or reading errors).

Conclusion

This C++ implementation effectively simulates a basic MapReduce framework. By using multithreading, synchronization, and inter-process communication, it demonstrates the working principles of distributed data processing systems. The project highlights key concepts like **parallelism**, **data shuffling**, and **aggregation**.