OPERATING SYSTEMS

PROJECT

Contributors:

Muhammad Shahzer (i22-2043)
Fasih-ur-Rehman (i22-1910)
Saim Nadeem (i22-1884)
Uzair Saddique (i22-6181)

DS-A

Date: 8th December, 2024

MapReduce Framework Implementation Using C++

Table of Contents

Introduction

The MapReduce framework is a programming paradigm designed to handle large-scale data processing by breaking it into smaller, parallelizable tasks. This project implements a basic MapReduce framework in C++ on a single machine, simulating distributed data processing using threads and inter-process communication.

The framework consists of three main phases: **Map**, **Shuffle**, and **Reduce**, where:

- The **Map phase** divides and processes input data into key-value pairs.
- The **Shuffle phase** groups data by keys.
- The **Reduce phase** aggregates the results to produce final output.

This report details the approach, code implementation, and results, with placeholders for flow and activity diagrams.

Approach And Design

Map Phase

In the Map phase:

1. Input text is divided into smaller **chunks**, each containing a portion of the input data.
2. **Threads** are created for parallel processing of chunks. Each thread:
    - Processes its chunk to extract individual words.
    - Emits key-value pairs in the form `("word", 1)`.
3. Key-value pairs are written to **named pipes** for inter-process communication.

Shuffle Phase

The Shuffle phase groups key-value pairs by key:

1. The Reducer reads key-value pairs from named pipes.
2. All occurrences of a particular key are combined into a list (e.g., `("blue", [1, 1, 1])`).
3. The system ensures that the data is grouped correctly before the Reduce phase.

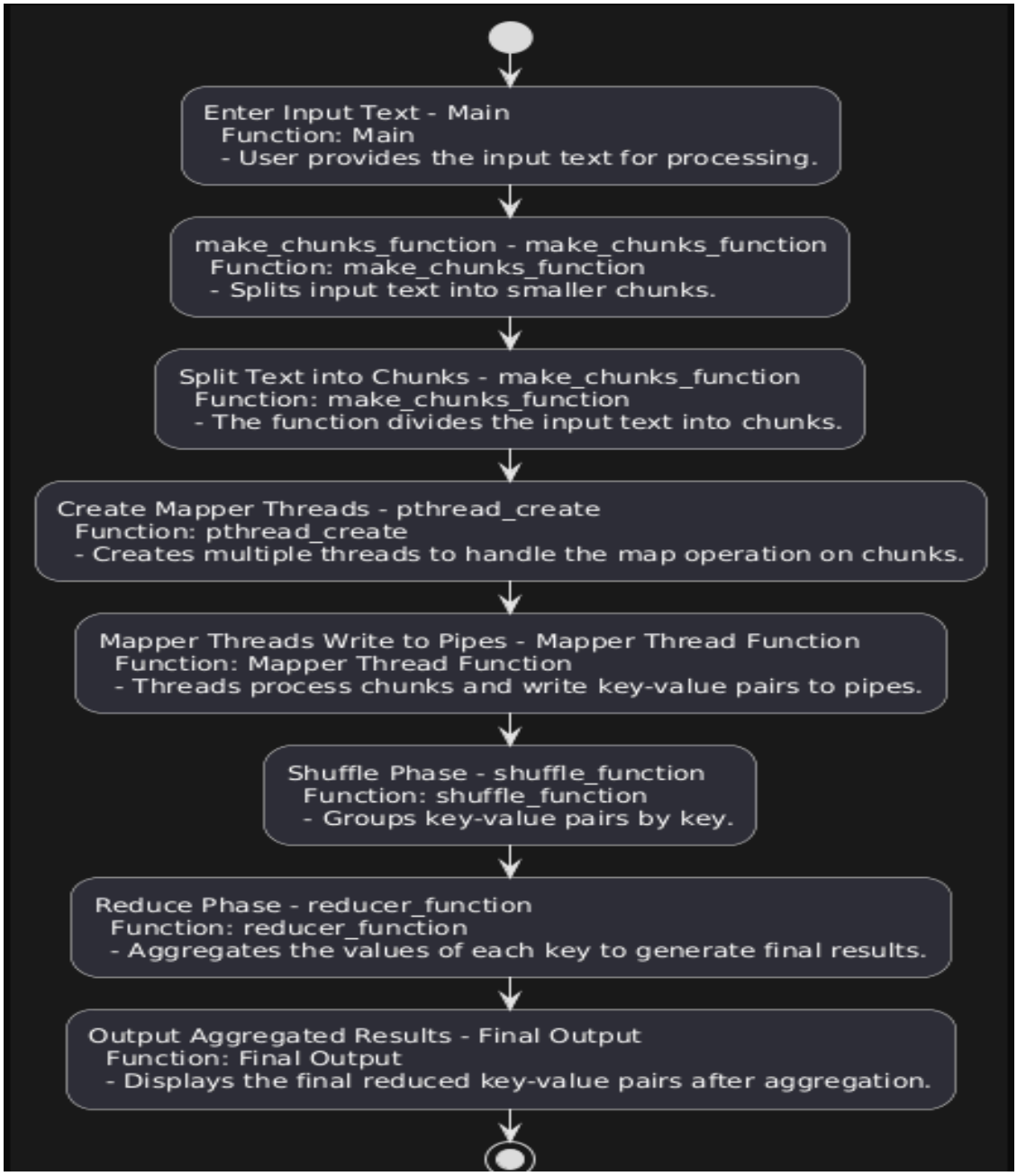
Reduce Phase

In the Reduce phase:

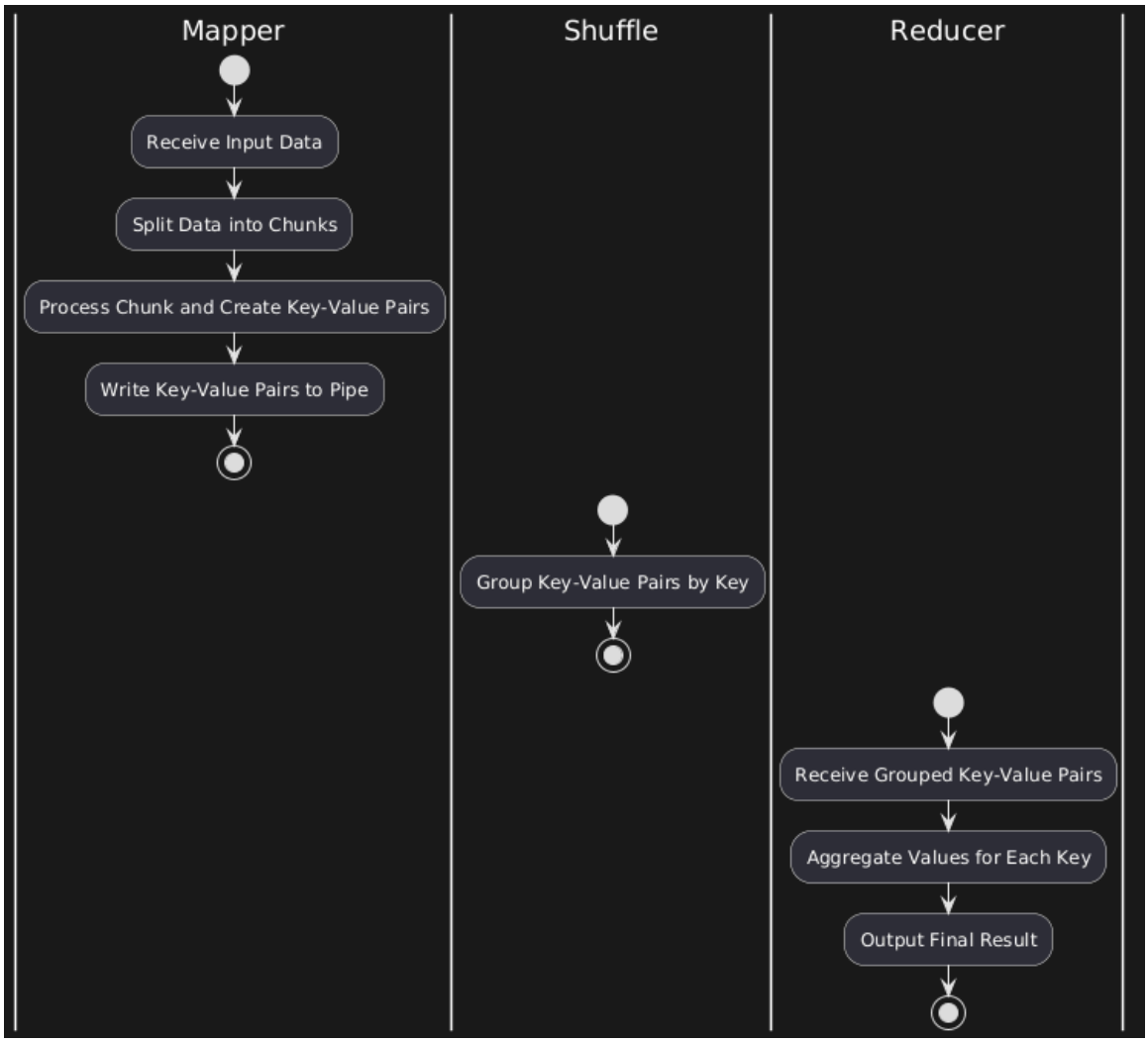
1. The Reducer processes grouped key-value pairs to aggregate values.
2. For each key, all associated values are summed up to produce the final count.
3. Results are displayed, representing the frequency of each word in the input text

Flow/Activity Diagram

High-Level Flow Diagram of MapReduce Framework



Activity Diagram for Mapper and Reducer Interaction



Code Explanation

Mapper Code

The Mapper is responsible for processing the input data and generating key-value pairs. Below are the key components explained:

1. `make_chunks_function`

This function divides the input text into smaller chunks based on the specified number of threads.

- Input:
  - `text`: The input string.
  - `chunk_number`: The number of chunks to divide the text into.
- Output:
  - An array of strings, each representing a chunk.

Logic:

1. Split the text into words.
2. Distribute words evenly across the specified number of chunks.

2. `thread_mapper_function`

This function is executed by each thread to process its assigned chunk.

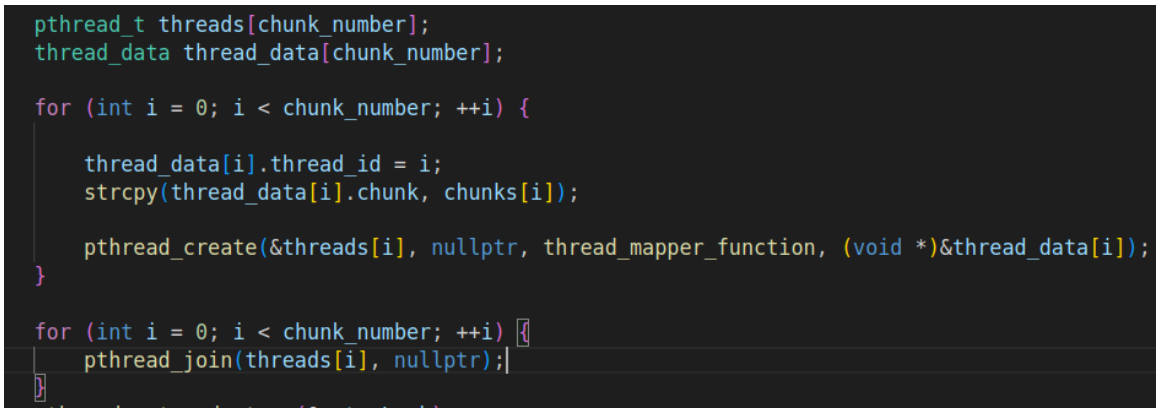
- Reads its chunk word by word.
- Emits key-value pairs in the form `("word", 1)`.
- Writes these pairs to a named pipe.

Synchronization:

- Uses `pthread_mutex` to prevent race conditions during output operations.
-

3. Thread Management

Threads are created for each chunk using `pthread_create`. After processing, the program waits for all threads to complete using `pthread_join`.



```
pthread_t threads[chunk_number];
thread_data thread_data[chunk_number];

for (int i = 0; i < chunk_number; ++i) {

    thread_data[i].thread_id = i;
    strcpy(thread_data[i].chunk, chunks[i]);

    pthread_create(&threads[i], nullptr, thread_mapper_function, (void *)&thread_data[i]);
}

for (int i = 0; i < chunk_number; ++i) {
    pthread_join(threads[i], nullptr);
}
```

Reducer Code

The Reducer reads data from named pipes, groups it by key, and aggregates the results.

1. `pipe_read`

Reads key-value pairs from named pipes created by Mapper threads.

- Input: Pipe name.
- Output: Parsed key-value pairs.
-

2. `shuffle_function`

Groups key-value pairs by key.

- Input: An array of key-value pairs.
- Output: A list of unique keys and their associated values.

Logic:

1. For each key, check if it exists in the unique keys list.
2. If found, append the value to the existing group; otherwise, create a new group.

3. `reducer_function`

Aggregates values for each unique key.

- Input: Unique keys and their grouped values.
- Output: The sum of values for each key.

```
for (int i = 0; i < unique_key_count; ++i) {
    int sum = 0;
    for (int j = 1; j <= grouped_value[i][0]; ++j) {
        sum += grouped_value[i][j];
    }
    cout << "Key: " << unique_key[i] << ", Aggregated Value: " << sum << endl;
}
```

Results

Below are example outputs demonstrating the execution of the MapReduce framework.

Example Input

```
red blue green yellow red green blue yellow blue green
```

Number of Chunks: 4

Example Mapper Output

```
shazer@shazer-HP-EliteBook-840-G5:~/Documents/os proj$ ./mapper
Enter the input text: red blue green yellow red green blue
yellow blue green
Enter the number of chunks: 4
Chunk 0: [red blue green]
Chunk 1: [yellow red green]
Chunk 2: [blue yellow]
Chunk 3: [blue green]
Mapper processing chunk: red blue green
Mapper processing chunk: yellow red green
Mapper processing chunk: blue yellow
Mapper processing chunk: blue green
Writing to pipe pipe_0: red 1
Writing to pipe pipe_0: blue 1
Writing to pipe pipe_0: green 1
Writing to pipe pipe_1: yellow 1
Writing to pipe pipe_1: red 1
```

```
Writing to pipe pipe_1: green 1
Writing to pipe pipe_2: blue 1
Writing to pipe pipe_2: yellow 1
Writing to pipe pipe_3: blue 1
Writing to pipe pipe_3: green 1
```

Example Reducer Output

```
--- Reading from Pipes ---

--- Shuffle Phase Results ---
Key: red, Value: 1 1
Key: blue, Value: 1 1 1
Key: green, Value: 1 1 1
Key: yellow, Value: 1 1

--- Reduce Phase ---
Key: red, Aggregated Value: 2
Key: blue, Aggregated Value: 3
Key: green, Aggregated Value: 3
Key: yellow, Aggregated Value: 2
```

Conclusion

This project successfully implements a basic MapReduce framework in C++ for single-machine execution. The results demonstrate:

- Efficient splitting of input data into chunks.
- Proper synchronization and inter-process communication.
- Accurate aggregation of results.

Future Scope

1. Extend the implementation to a distributed system with multiple nodes.
2. Optimize for handling larger datasets by using more advanced data structures.
3. Incorporate fault tolerance for thread and process failures.