Technical Report: Implementation of a Multithreaded MapReduce Word Count System

Report

Aaqib Ahmed Nazir I22-1920

Arhum Khan I22-1967

Ammar Khasif I22-1968

DS-D

Introduction

The MapReduce paradigm has revolutionized parallel computing by allowing complex data processing tasks to be divided into smaller, independent tasks that can be executed concurrently. This project implements a simplified version of the MapReduce model to count the occurrences of words in a text. The solution is designed with a focus on efficient memory use, thread synchronization, and high performance.This document provides a detailed explanation of the approach, system design, key findings, and steps to run the project.
Github repo

Overview

The implementation consists of three major phases: **Map, Shuffle, and Reduce**, designed to process input data in parallel using threads. Each phase performs a specific task:

1. **Input Processing:** The raw input text is sanitized, split into individual words, and distributed to mapper threads.
2. **Map Phase:** Mapper threads process chunks of text, creating key-value pairs for each word and storing them in a shared data structure.
3. **Shuffle Phase:** The shuffle phase groups and organizes key-value pairs, preparing them for reduction.
4. **Reduce Phase:** Reducer threads aggregate the word counts to produce the final output.

The system uses **POSIX threads (pthreads)** for concurrency, **mutexes** for shared data protection, and **semaphores** for coordinating transitions between phases.
Key features of this implementation include:

- A modular design that ensures reusability and scalability.
- Thread-safe handling of shared data.
- Memory-efficient operations with systematic cleanup to avoid leaks.

System Design

System Workflow

1. **Input Processing**:

    • The input text is split into words and allocated to mapper threads.

    • Words are sanitized by removing special characters and converting to lowercase.

    • Chunks of words are assigned to mappers for processing.

2. **Mapping Phase**:

    • Each mapper processes its assigned chunk, creating key-value pairs (word, count) for each word.

    • Intermediate data is stored in a shared structure, protected by mutexes to ensure thread safety.

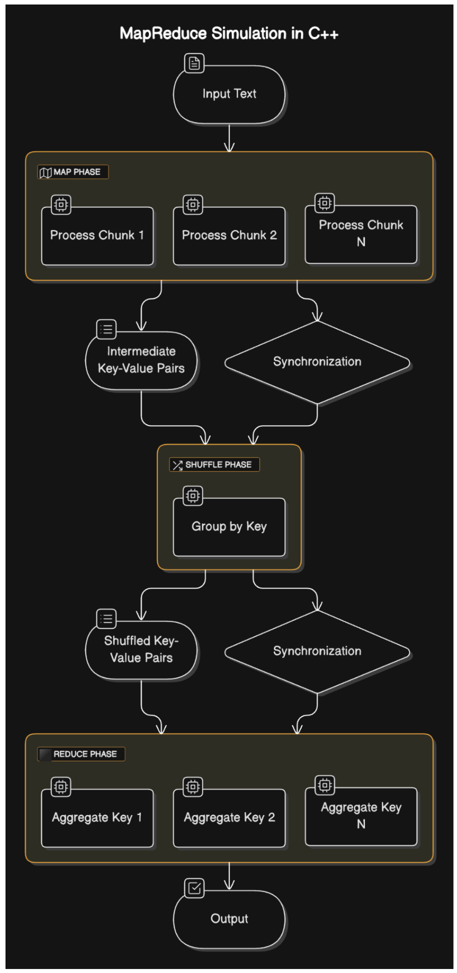
3. **Shuffling Phase**:

    - Once all mappers finish, a dedicated thread performs the shuffle operation.
    - It groups words, organizes them alphabetically, and prepares the data for reduction.

4. **Reducing Phase**:

    - Each reducer thread handles one or more unique keys (words).
    - The reducer aggregates counts for its assigned key and generates the final output.

Diagram and Explanation



Explanation:

- **Input Processing:** The raw input text is cleaned, converted into words, and distributed to mapper threads.
- **Mapping Phase:** Parallel threads process text chunks independently and generate intermediate key-value pairs for each word.
- **Shuffling Phase:** A centralized process sorts and groups the key-value pairs for easier reduction.
- **Reducing Phase:** Multiple threads work on grouped keys to calculate the final counts.
- **Final Output:** The aggregated word frequencies are displayed in a user-friendly format.

Code Explanation

The solution is implemented across four core files: **main.cpp**, **functions.cpp**, **functions.h**, and **testcases.cpp**. The key processes are as follows:

1. Input Processing

- **Objective:** Prepare the input text for mappers.
- **Process:**
  - Input is tokenized into words.
  - Non-alphanumeric characters are removed, and words are converted to lowercase.
  - Words are distributed evenly across mapper threads.

2. Mapping Phase

- **Objective:** Generate key-value pairs for each word.
- **Process:**
  - Each mapper thread processes a chunk of text.
  - Words are checked against the intermediate data for duplicates. If found, their count is incremented; otherwise, a new key-value pair is created.
- **Synchronization:** Mutex locks ensure thread-safe access to shared data.

3. Shuffling Phase

- **Objective:** Organize key-value pairs for reducers.
- **Process:**
  - After all mappers complete, the shuffle thread groups words and sorts them alphabetically.
- **Output:** Key-value pairs are prepared for aggregation in the reduce phase.

4. Reducing Phase

- **Objective:** Aggregate counts for each unique word.
- **Process:**
  - Each reducer thread is assigned a unique word.
  - Counts from all occurrences of the word in the intermediate data are summed.
  - Results are written to the final output.

Types of Test Cases

The implementation was rigorously tested using a variety of scenarios, including:

1. **Basic Tests**: Simple inputs with evenly distributed word counts.
2. **Edge Cases**: Single-word inputs, large texts, and empty inputs.
3. **Stress Tests**: Extremely large input sizes to evaluate performance under high load.
4. **Special Cases**: Inputs with special characters, varying cases, and mixed spacing.

Key Findings

- **Performance:** The solution demonstrated efficient parallel processing, with noticeable improvements in execution time for large inputs.
- **Accuracy:** All words were correctly counted, including those with special characters and mixed cases (after sanitization).
- **Scalability:** The multithreaded design performed well under stress, processing thousands of words efficiently.
- **Synchronization:** Mutexes and semaphores ensured thread safety, with no data races or inconsistencies.
- **Memory Management:** Proper allocation and deallocation prevented memory leaks, even in edge cases.

How to Run the Project

To build and run the project, the following steps are required:

Steps to Run:

1. **To build and run the main program**:
   make main

2. **To build and run the test cases**:
   make test

3. **To compile the program without running it**:
   make Or make all

4. **To clean up compiled files**:
   make clean

Conclusion

This MapReduce implementation successfully showcases the power of parallel processing in handling large-scale data. The use of multithreading, thread-safe mechanisms, and efficient data structures enabled a robust and scalable solution for the word count problem.