UNIVERSITY OF SCIENCE AND TECHNOLOGY OF HANOI
DEPARTMENT OF INFORMATION AND COMMUNICATION TECHNOLOGY

Practical Work 4
Word Count using Simulated MapReduce

Author:
Nguyen Tuan Dung
Student ID: 23BI14113

December 5, 2025

Contents

1 Introduction

This report details the implementation of the classic Word Count problem using a simulated MapReduce framework, built with C++ for the core logic. Since no native C++ MapReduce framework was used, the execution environment (Splitting, Shuffling, Grouping) is simulated using shell commands and standard I/O redirection, allowing the focus to remain strictly on the parallelizable logic of the **Mapper** and **Reducer** functions.

2 MapReduce Implementation Choice

2.1 Framework Rationale

The requirement was to use C/C++ for implementation despite the lack of C++ MapReduce framework (unlike Hadoop/Spark for Java/Scala/Python). Therefore, the chosen implementation is a **Simulated Streaming MapReduce**.

- **C++ for Core Logic:** C++ was chosen for its performance, aligning with the preference for low-level execution efficiency in large-scale data tasks.
- **Streaming Model:** The C++ executable (`word_count`) is designed to be executable in two modes (`mapper` or `reducer`). It reads input from `stdin` and writes output to `stdout`.
- **Framework Simulation:** The complex tasks of **splitting**, **shuffling**, and **sorting/-grouping** are offloaded to standard operating system utilities (`split`, `sort`) or a minimal shell script, mimicking the role of the MapReduce Master/Scheduler.

3 Mapper and Reducer Logic

The Word Count problem is solved by defining the key-value transformation across the two phases.

3.1 Mapper Logic

The Mapper takes lines of text (the input split) and emits an intermediate key-value pair for every word found.

Table 1: Mapper Input/Output Specification

Input Type	Mapper Task	Intermediate Output
Text Line	Tokenize the line into words. Normalize (lowercase, remove punctuation).	`<word, 1>`

3.2 Reducer Logic

The Reducer receives the globally sorted and grouped data, ensuring that all occurrences of the same word are fed sequentially.

Table 2: Reducer Input/Output Specification

Intermediate Input	Reducer Task	Final Output
`<word, list(1, 1, ...)>`	Iterate through the list of ones. Sum the counts for the current word.	`<word, total_count>`

4 System Organization and Workflow

The organization uses the file system and standard I/O redirection to orchestrate the parallel execution.

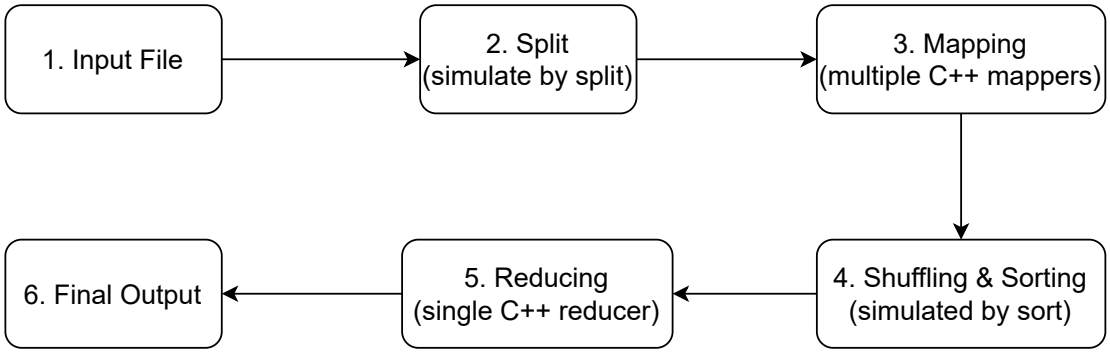


Figure 1: Simulated MapReduce Workflow. The shell environment acts as the Master node managing the data flow between the C++ Map and Reduce executables.

4.1 Who Does What

- **The C++ Executable (`word_count`):**
  - **Mapper Role:** Executes pure, local word tokenization and emits raw key-value pairs (`word 1`).
  - **Reducer Role:** Executes pure, local aggregation by summing sequential keys and emitting the final count.
- **The Operating System/Shell (Simulated Master/Framework):**
  - **Splitting:** A shell command (`split`) logically divides the input file.
  - **Shuffling & Sorting:** The `sort` utility is used to globally sort the Mappers' outputs by key, which is the crucial step that feeds the Reducer correctly.
  - **Grouping:** The Reducer's C++ logic implicitly groups the values because the input stream is guaranteed to be sorted by key.

5 Implementation Snippets (C++ Core Logic)

5.1 Mapper Snippet

The Mapper focuses on simple normalization.

Listing 1: C++ Mapper Core Logic (Tokenization and Emission)

```
void run_mapper() {
    string line;
    while (getline(cin, line)) {
        stringstream ss(line);
        string word;
        while (ss >> word) {
            string normalized_word = normalize(word);
            if (!normalized_word.empty()) {
                // Output: <word>\t<1>
                cout << normalized_word << "\t" << 1 << endl;
            }
        }
```

```
    }
}
```

5.2 Reducer Snippet

The Reducer relies entirely on the fact that the input stream is sorted (‘word‘ by ‘word‘).

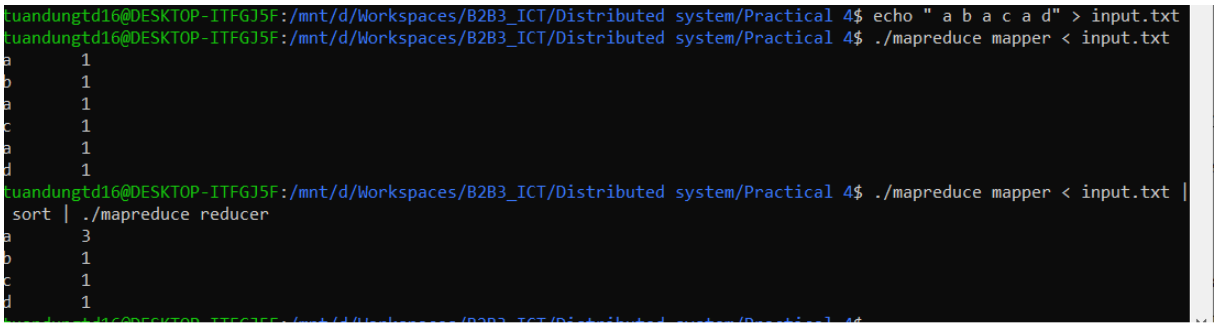
Listing 2: C++ Reducer Core Logic (Aggregation)

```cpp
void run_reducer() {
    string line;
    string current_word = "";
    int current_count = 0;

    // Input is guaranteed to be sorted: <word> <1>
    while (getline(cin, line)) {
        // ... (Parse word and count) ...

        if (word != current_word) {
            // Output result for the previous word group
            if (!current_word.empty()) {
                cout << current_word << "\t" << current_count << endl;
            }
            // Start new group
            current_word = word;
            current_count = count;
        } else {
            // Aggregate: Sum the count for the current group
            current_count += count;
        }
    }
    // Output the final group
    if (!current_word.empty()) {
        cout << current_word << "\t" << current_count << endl;
    }
}
```

6 Results



7 Conclusion

By decoupling the core logic (C++ executables) from the framework logic (OS utilities), this simulation successfully demonstrates the principles of MapReduce. The system is inherently parallelizable: adding more Mapper instances (and configuring the shell pipeline accordingly) allows the processing of larger input files efficiently, fulfilling the core promise of the MapReduce model.