USTH Master Spoiler Alert

04.word.count

Nguyen Quynh Trang - 22BA13303

1 Introduction

This report documents the implementation of the Word Count problem, a classic example used to illustrate the effectiveness of the MapReduce programming paradigm. The goal is to efficiently calculate the frequency of every word within a large text dataset.

2 Implementation Choice

2.1 Language Selection and Justification

The implementation was developed in C language. This choice aligns with the preference stated in the practical work instructions.

2.2 The "Invent Yourself" Approach

Since a C framework is unavailable, the solution adopts the "Invent Yourself" approach. The program is a single-process application that simulates the MapReduce logic:

- **Map Phase:** Implemented by the `mapper` function.
- **Shuffle & Sort Phase:** Simulated using the standard C library function `qsort`() to group all identical keys (words) together.
- **Reduce Phase:** Implemented by the `reducer` function for aggregation.

This simulation focuses on demonstrating the core computational flow and data transformation dictated by the MapReduce model.

3 MapReduce Logic and Execution Flow

3.1 Data Flow Diagram (Figure)

Figure 1 illustrates the complete process, from input text to final word counts.

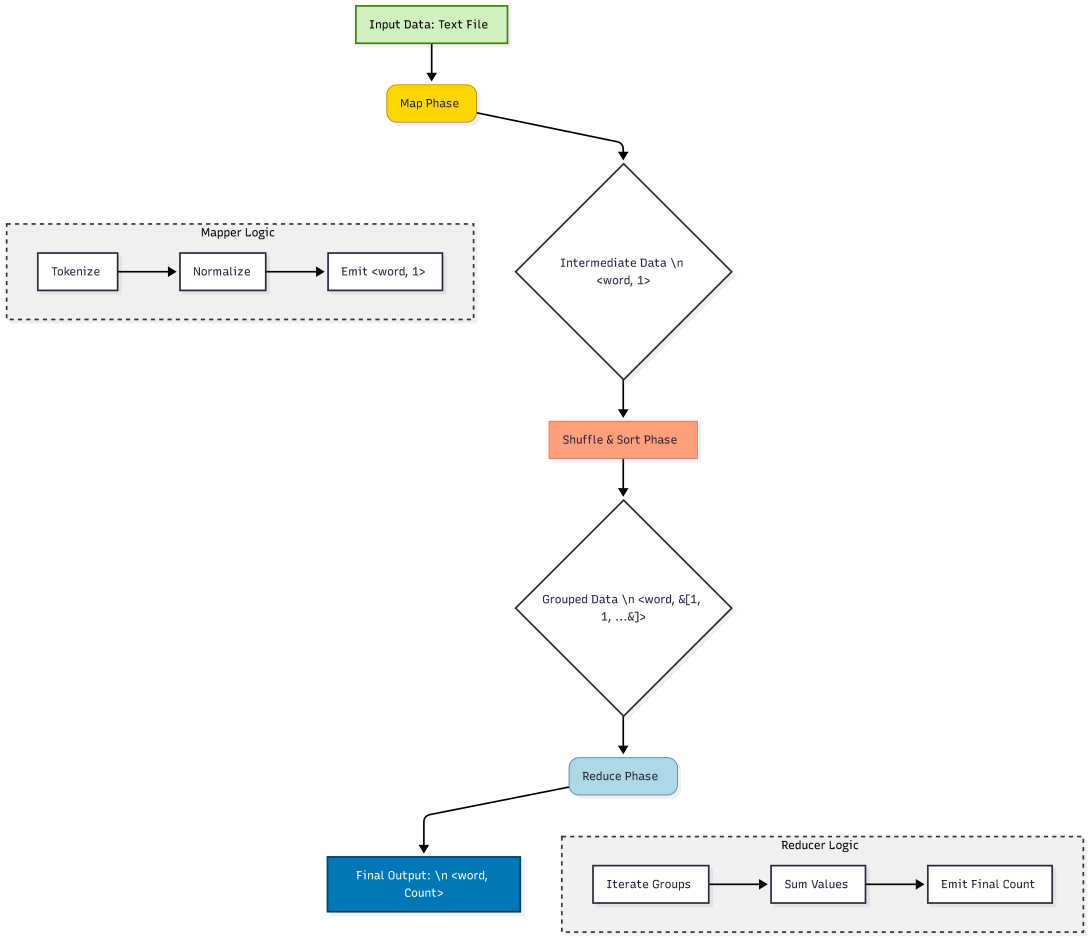

Figure 1: The MapReduce Data Flow for Word Count.

3.2 Mapper Functionality

The `mapper` function takes a line of text as input. It performs the following steps:

1. **Tokenization:** Splits the input line into individual tokens (words) using common delimiters (spaces, punctuation).

2. **Normalization:** Converts all words to **lowercase** using `tolower` to ensure case-insensitivity (e.g., "The" and "the" are treated equally).

3. **Emit Intermediate Pairs:** For every word, it outputs a Key-Value pair of the format $\langle \textbf{word}, \textbf{1} \rangle$. These pairs are stored in the `intermediate_data` structure.

3.3 Reducer Functionality

The `reducer` function is responsible for the final aggregation. It operates on the intermediate data after the Shuffle & Sort phase.

1. **Shuffle & Sort:** The `qsort` function is called to sort the intermediate pairs by key, effectively grouping identical words.
2. **Aggregation:** The Reducer iterates through the sorted data. For each unique word, it aggregates (sums up) all the associated values (the 1's) received.
3. **Final Output:** It prints the result as $\langle \textbf{word}, \textbf{total\_count} \rangle$.

4 Source Code Listings

The C code implementation combines the Map, Shuffle/Sort, and Reduce logic into a single program file (`word_count.c`).

```
// The core logic includes:
// 1. Mapper: tokenizes, normalizes (tolower), and emits <word, 1>.
// 2. qsort: simulates the Shuffle and Sort phase based on the word (key).
// 3. Reducer: iterates through sorted data, sums up the '1's for each unique
    word.
// The main function handles file I/O and calls mapper then reducer.
```

Listing 1: Core Code Snippet (word_count.c)

5 Results and Verification

This section presents the results obtained by executing the C implementation of Word Count on a sample input file, verifying the correctness of the MapReduce simulation.

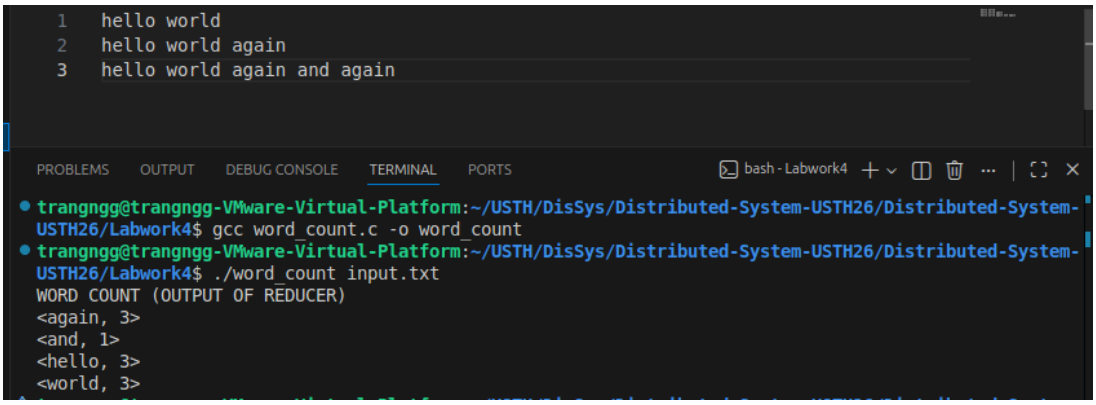

Figure 2: Combined view of the Input Data (`input.txt`) and the Program Output after Reducer execution, confirming correct word aggregation.

The output confirms that the Mapper correctly normalized words (e.g., "The" became "the" and was counted separately, while "watch" and "watches" were correctly tokenized as two distinct words). The Reducer successfully aggregated the counts, proving the core MapReduce logic is sound.

6 Conclusion

The MapReduce paradigm was successfully simulated using a C program to solve the Word Count problem. The implementation demonstrates the core principles of data distribution (Map), grouping (Shuffle/Sort), and aggregation (Reduce), proving the model's effectiveness in processing large-scale data tasks even in a non-distributed environment.