Multithreaded Word Count Using Mapper–Reducer Style

Hoang Minh Quan
Student ID: 23BI14371

December 5, 2025

Abstract

This report presents a small C++ application that counts word frequencies in a text file using a mapper–reducer style design. The program reads an input file, splits the text into chunks that are processed in parallel by mapper threads, and then combines the intermediate results with reducer threads before writing the global word counts to an output file.

Contents

1 Introduction

Counting how often each word appears in a text is a simple but representative problem for data-processing frameworks. In this assignment, we implement a standalone C++ program that performs word counting in parallel on a single machine. The program is inspired by the MapReduce model: we use several *mapper* threads to create local word-count maps and several *reducer* threads to merge these partial results into a final map.

The implementation uses the C++ standard library, threads, and `std::unordered_map` to store word frequencies. Input and output are handled via standard text files.

2 Problem Description

The goal is to build a command-line tool with the following behaviour:

- read a text file given by the user;
- normalise words (convert to lowercase and remove non-alphabetic characters);
- count how many times each distinct word occurs;
- allow the user to configure how many mapper and reducer threads are used;
- store the word counts in a separate output file.

The provided example input file contains the sentence

```
why do programmers prefer dark mode because light attracts bugs
```

and the expected output lists each word once together with its frequency, for example:

```
attracts 1
because 1
bugs 1
dark 1
do 1
light 1
mode 1
prefer 1
programmers 1
why 1
```

3 System Overview

3.1 High-level Design

The program follows four main stages:

1. **Load file:** the entire input file is read into a single string.
2. **Map phase:** the text is partitioned into chunks, each processed by a mapper thread producing a local `(word, count)` map.
3. **Shuffle phase:** intermediate pairs are grouped so that all counts of the same word are sent to the same reducer.
4. **Reduce phase:** reducer threads accumulate final word counts and the result is written to the output file.

3.2 Illustration

Figure 1 illustrates the architecture of the application. A single input text is transformed into several mapper chunks, aggregated via reducer threads, and finally written as a sorted list of words and frequencies.

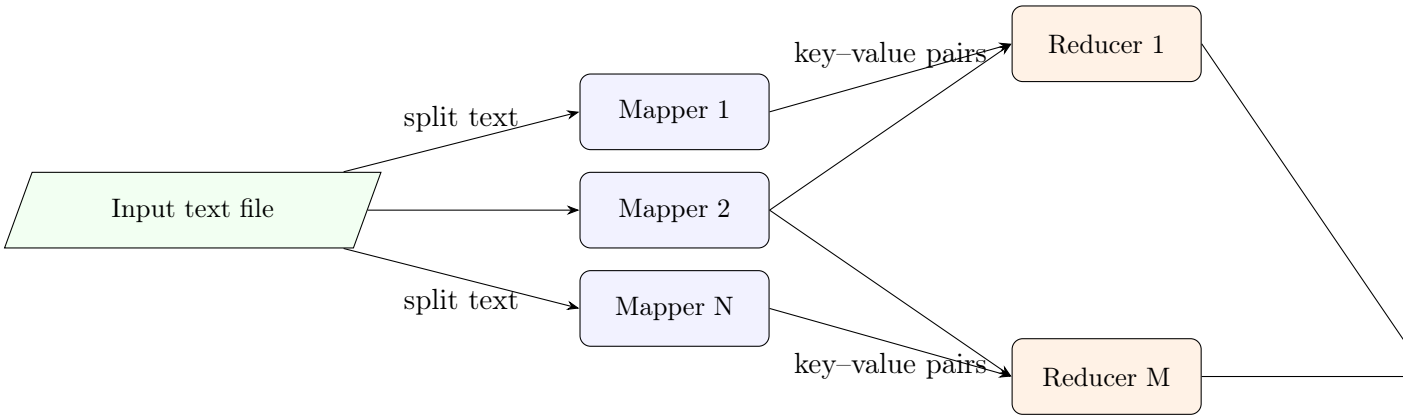

Figure 1: Mapper–reducer style architecture for the word-count tool.

4 Implementation Details

4.1 Word Normalisation and Splitting

The function `normalize` converts characters to lowercase and removes anything that is not an alphabetic letter. A second helper function, `split_words`, walks through the input text, extracts sequences of non-whitespace characters, normalises them, and returns a vector of clean words.

4.2 Mapper Threads

Each mapper thread receives a substring ("chunk") of the original text, calls `split_words` on that chunk, then builds a local `unordered_map<std::string,int>` that counts occurrences of each word in the chunk.

To avoid cutting words in half, the code adjusts the chunk boundary to the next whitespace character before launching the mapper.

4.3 Reducer Threads

After all mapper threads have finished, the program distributes the intermediate `(word, count)` pairs to reducers. A simple hash partitioning scheme is used:

$$\text{reducer\_id} = \text{hash(word)} \bmod \text{num\_reducers}. \tag{1}$$

Each reducer accumulates the counts it receives into its own `unordered_map`. Finally, these per-reducer maps are merged into a single global map.

4.4 Source Code Listing

Listing 1 shows the full implementation of the program.

```
#include <algorithm>
#include <cctype>
#include <fstream>
#include <iostream>
```

```cpp
#include <mutex>
#include <string>
#include <thread>
#include <unordered_map>
#include <utility>
#include <vector>

using MapperOutput = std::unordered_map<std::string, int>;
using ReducerOutput = std::unordered_map<std::string, int>;

std::mutex io_mutex;

std::string normalize(const std::string &w) {
    std::string res;
    for (char c : w) {
        if (std::isalpha(static_cast<unsigned char>(c))) {
            res.push_back(static_cast<char>(std::tolower(c)));
        }
    }
    return res;
}

std::vector<std::string> split_words(const std::string &text) {
    std::vector<std::string> words;
    std::string current;
    for (char c : text) {
        if (std::isspace(static_cast<unsigned char>(c))) {
            if (!current.empty()) {
                std::string norm = normalize(current);
                if (!norm.empty()) words.push_back(norm);
                current.clear();
            }
        } else {
            current.push_back(c);
        }
    }
    if (!current.empty()) {
        std::string norm = normalize(current);
        if (!norm.empty()) words.push_back(norm);
    }
    return words;
}

void mapper_worker(const std::string &chunk, MapperOutput &out_map) {
    auto words = split_words(chunk);
    for (const auto &w : words) {
        ++out_map[w];
    }
}

void reducer_worker(const std::vector<std::pair<std::string, int>> &
    pairs,
                    ReducerOutput &out_map) {
    for (const auto &p : pairs) {
        out_map[p.first] += p.second;
    }
}

```

```cpp
int main(int argc, char **argv) {
    if (argc < 3) {
        std::cerr << "Usage: " << argv[0]
                  << " <input_file> <output_file> [num_mappers] [num_reducers]\n";
        return 1;
    }

    std::string input_file = argv[1];
    std::string output_file = argv[2];
    int num_mappers = (argc >= 4) ? std::stoi(argv[3]) : 4;
    int num_reducers = (argc >= 5) ? std::stoi(argv[4]) : 2;

    if (num_mappers <= 0 || num_reducers <= 0) {
        std::cerr << "Number of mappers and reducers must be > 0\n";
        return 1;
    }

    std::ifstream in(input_file, std::ios::binary);
    if (!in.is_open()) {
        std::cerr << "Cannot open input file: " << input_file << "\n";
        return 1;
    }
    std::string text((std::istreambuf_iterator<char>(in)),
                     std::istreambuf_iterator<char>());
    in.close();

    if (text.empty()) {
        std::cerr << "Input file is empty.\n";
        return 1;
    }

    std::vector<std::thread> mapper_threads;
    std::vector<MapperOutput> mapper_outputs(num_mappers);

    size_t chunk_size = text.size() / num_mappers;
    size_t start = 0;

    for (int i = 0; i < num_mappers; ++i) {
        size_t end = (i == num_mappers - 1) ? text.size() : start + chunk_size;

        if (end < text.size()) {
            while (end < text.size() &&
                   !std::isspace(static_cast<unsigned char>(text[end]))) {
                ++end;
            }
        }

        std::string chunk = text.substr(start, end - start);
        mapper_threads.emplace_back(mapper_worker, chunk,
                                    std::ref(mapper_outputs[i]));
        start = end;
    }

    for (auto &t : mapper_threads) t.join();

```

```
    std::vector<std::vector<std::pair<std::string, int>>> reducer_inputs(
        num_reducers);

    for (const auto &m_out : mapper_outputs) {
        for (const auto &kv : m_out) {
            const std::string &word = kv.first;
            int count = kv.second;
            std::size_t h = std::hash<std::string>{}(word);
            int rid = static_cast<int>(h % num_reducers);
            reducer_inputs[rid].emplace_back(word, count);
        }
    }

    std::vector<std::thread> reducer_threads;
    std::vector<ReducerOutput> reducer_outputs(num_reducers);

    for (int r = 0; r < num_reducers; ++r) {
        reducer_threads.emplace_back(reducer_worker,
                                     std::cref(reducer_inputs[r]),
                                     std::ref(reducer_outputs[r]));
    }

    for (auto &t : reducer_threads) t.join();

    std::unordered_map<std::string, int> final_counts;
    for (const auto &r_out : reducer_outputs) {
        for (const auto &kv : r_out) {
            final_counts[kv.first] += kv.second;
        }
    }

    std::vector<std::pair<std::string, int>> sorted(final_counts.begin(),
                                                    final_counts.end());
    std::sort(sorted.begin(), sorted.end(),
              [](auto &a, auto &b) { return a.first < b.first; });

    std::ofstream out(output_file);
    if (!out.is_open()) {
        std::cerr << "Cannot open output file: " << output_file << "\n";
        return 1;
    }

    for (const auto &kv : sorted) {
        out << kv.first << " " << kv.second << "\n";
    }
    out.close();

    std::cout << "Word count written to: " << output_file << "\n";
    return 0;
}
```

Listing 1: C++ implementation of the multithreaded word-count tool

5 Example Run

With the example sentence given earlier, the program can be compiled and executed as:

```
mpic++ is not required here; we simply use g++ or clang++:

g++ -std=c++17 -pthread wordcount.cpp -o wordcount
./wordcount input.txt output.txt 4 2
```

The resulting `output.txt` contains one line per word and its frequency, as shown previously. For this short input, each word appears exactly once.

6 Conclusion

This small project shows how the mapper–reducer idea can be implemented using only the C++ standard library. Even though the example input is short, the design naturally scales to larger text files: more mapper and reducer threads can be enabled to use the available CPU cores. Possible extensions include:

- adding command-line options to filter stop words;
- supporting case-sensitive or case-insensitive counting modes;
- exporting the results in formats such as CSV or JSON for further analysis.