Practical Work 4: Word Count

Nguyen Hoang Long - 23Bi14270

1 Goal

The objective of this practical work is to implement a Word Count application using a MapReduce approach. A new directory named `WordCount` is created and a simple MapReduce framework is implemented in C++ without relying on any external library.

The system must:

- read an input text file;
- split the processing into a *map* and a *reduce* phase;
- count the number of occurrences of each distinct word;
- output the result as `word count` pairs to a file.

2 Choice of MapReduce Implementation

The implementation is written in C++17 and uses:

- `std::thread` to simulate parallel mappers and reducers;
- `std::unordered_map` to store intermediate key–value pairs;
- `std::hash<std::string>` to assign keys to reducers.

3 Design of the MapReduce Service

Map Phase

The input file is read entirely into memory and partitioned into *chunks*. Each chunk is processed by a mapper thread. A mapper:

- splits the text chunk into words;
- normalizes each word (lowercasing, removing punctuation);
- emits intermediate key–value pairs $(word, 1)$;
- stores results in a local `unordered_map`.

Shuffle Phase

All intermediate results are then redistributed to reducers:

- each word is assigned to a reducer index using the hash of the word modulo the number of reducers;
- for each reducer, a vector of (*word*, *count*) pairs is built.

Reduce Phase

Each reducer thread receives a list of pairs and:

- aggregates counts for each word;
- produces a local map from word to total count;
- the main thread finally merges all reducer outputs into a single final map of word counts.

Design Figure

MapReduce Word Count Design

Input file → split into chunks.
Each chunk → processed by a Mapper thread → local word counts.
Shuffle step groups words by hash(word) mod #reducers.
Each Reducer thread aggregates counts for its assigned words.
Final stage merges reducer outputs and writes `word count` pairs.

Figure 1: Overall design of the MapReduce Word Count system.

4 System Organization

The system consists of a single executable `wordcount` built from `wordcount.cpp`. The program is executed as:

```
./wordcount input.txt output.txt [num_mappers] [num_reducers]
```

- **Input module**: reads the entire file into a string.
- **Map module**: spawns mapper threads and stores per-mapper `unordered_map` objects.
- **Shuffle module**: redistributes intermediate data into a vector of lists, one per reducer.
- **Reduce module**: spawns reducer threads and aggregates counts.
- **Output module**: sorts the final map by word and writes the result to the output file.

5 Implementation of the File Transfer

Mapper and Reducer Implementation

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

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;
    }
}
```

Main Control Flow

```cpp
int main(int argc, char **argv) {
    std::string input_file = argv[1];
    std::string output_file = argv[2];
    int num_mappers = std::stoi(argv[3]);
    int num_reducers = std::stoi(argv[4]);

    std::ifstream in(input_file);
    std::string text((std::istreambuf_iterator<char>(in)),
                     std::istreambuf_iterator<char>());
    in.close();

    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;
        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) {
            size_t h = std::hash<std::string>{}(kv.first);
            int rid = static_cast<int>(h % num_reducers);
            reducer_inputs[rid].emplace_back(kv.first, kv.second);
        }
    }

    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());

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

    return 0;
}
```

The full implementation is contained in `wordcount.cpp` and includes word normalization, command-line parsing, and error handling.

6 Conclusion

This practical work demonstrates how a simple MapReduce-style framework can be implemented directly in C++ using threads, hash maps, and basic data structures. The Word Count example shows:

- how to separate the application into map, shuffle, and reduce phases;

- how to use multiple threads to process different chunks of the input text in parallel;
- how to design a simple key-based partitioning scheme for reducers.

The same structure can be extended to other problems that can be expressed in the MapReduce model, such as inverted index construction or log aggregation.