



University Of Science And
Technology
Of Hanoi
ICT Department

LAB REPORT

Title: Practical Work 4 – MapReduce Word Count

Subject: Distributed Systems (DS2026)
Student: Le Nam Anh
StudentID: 23BI14029
Major: Cyber Security

1 Introduction

The objective of this practical work is to solve the classic **Word Count** problem using the MapReduce programming model. Instead of using existing frameworks like Hadoop, I "invent" a custom MapReduce framework using **C++**, as suggested in the course lectures.

The system is implemented to run on a **Linux environment**, utilizing multi-threading to simulate distributed worker nodes processing data in parallel.

2 Implementation Choice

To fulfill the requirement of "inventing" a MapReduce framework from scratch, I chose to implement the system using **C++ (Standard Library)** on a **Linux environment** with several advantages:

- **Manual Concurrency Control:** I utilized `std::thread` to manually simulate independent worker nodes. This allows for a granular demonstration of how parallel processing works at a system level.
- **Performance & Memory Management:** C++ offers superior performance for CPU-bound tasks like text processing. Using `std::map` (Red-Black Tree) for intermediate data storage.
- **Environment Suitability:** The system was developed on Linux to leverage the native GCC compiler and its robust support for POSIX-style threading (`-pthread`), providing a stable environment for multi-threaded execution.

3 System Design

The system follows a **Shared-Nothing** architecture during the Map phase to avoid race conditions, followed by a synchronization point for the Reduce phase.

3.1 How the Mapper works

The **Mapper** logic is encapsulated in the `map_function`. It operates on a specific "chunk" of the input file (a large string) assigned by the Master.

- **Input:** A raw text chunk.
- **Process:**
 - The text is tokenized into individual words using `stringstream`.
 - Each word is normalized (punctuation removed, converted to lowercase) to ensure consistency (e.g., "Hello," becomes "hello").
 - **Local Aggregation:** Instead of emitting a stream of `<word, 1>` pairs immediately, the Mapper aggregates counts into a **thread-local std::map<string, int>**.
- **Output:** An intermediate map containing word counts specific to that chunk (e.g., `{"hello": 2, "world": 1}`). This strategy significantly reduces the memory overhead compared to storing a list of all key-value pairs.

3.2 How the Reducer works

The **Reducer** logic is implemented in the `reduce_function` and executed by the Master thread after all Mappers have finished.

- **Input:** A vector containing all the local maps produced by the worker threads.
- **Process:**
 - The Reducer iterates through each local map.
 - For each key-value pair `<word, count>`, it adds the count to the corresponding entry in the **Global Result Map**.
 - Since the Map phase is complete, the Reducer can safely write to the global map without needing mutex locks for every single addition.
- **Output:** A final map containing the total frequency of every unique word in the input file.

Figure 1: System Architecture Diagram

4 Implementation Details

The core logic is encapsulated in `wordcount.cpp`.

Listing 1: MapReduce Implementation in C++

```
1 #include <iostream>
2 #include <fstream>
3 #include <string>
4 #include <vector>
5 #include <map>
6 #include <thread>
7 #include <sstream>
8 #include <algorithm>
9 #include <mutex>
10
11 using namespace std;
12
13 const int NUM_THREADS = 4;
14
15 string clean_word(const string& w) {
16     string res = "";
17     for (char c : w) {
18         if (isalnum(c)) res += tolower(c);
19     }
20     return res;
21 }
22
23 //MAPPER
24 void map_function(const string& text_chunk, map<string, int>&
25   local_result) {
26     stringstream ss(text_chunk);
27     string word;
28     while (ss >> word) {
```

```

28     string cleaned = clean_word(word);
29     if (!cleaned.empty()) local_result[cleaned]++;
30 }
31 }
32
33 //REDUCER
34 void reduce_function(const vector<map<string, int>>& all_maps, map<
35   string, int>& final_result) {
36   for (const auto& local_map : all_maps) {
37     for (const auto& pair : local_map) {
38       final_result[pair.first] += pair.second;
39     }
40   }
41
42 int main(int argc, char* argv[]) {
43   if (argc < 2) {
44     cout << "Usage: ./wordcount <filename>" << endl;
45     return 1;
46   }
47
48   //INPUT
49   ifstream file(argv[1]);
50   if (!file.is_open()) return 1;
51   string content((istreambuf_iterator<char>(file)),
52   istreambuf_iterator<char>());
53   file.close();
54   cout << "[Master] Read file size: " << content.length() << " bytes."
55   << endl;
56
57   //SPLIT
58   vector<string> chunks(NUM_THREADS);
59   size_t chunk_size = content.length() / NUM_THREADS;
60   for (int i = 0; i < NUM_THREADS; ++i) {
61     chunks[i] = (i == NUM_THREADS - 1) ?
62       content.substr(i * chunk_size) :
63       content.substr(i * chunk_size, chunk_size);
64   }
65
66   //MAP
67   cout << "[Master] Starting " << NUM_THREADS << " threads..." << endl
68   ;
69   vector<thread> threads;
70   vector<map<string, int>> intermediate_results(NUM_THREADS);
71
72   for (int i = 0; i < NUM_THREADS; ++i) {
73     threads.push_back(thread(map_function, ref(chunks[i]), ref(
74     intermediate_results[i])));
75   }
76
77   for (auto& t : threads) t.join();
78   cout << "[Master] Map phase complete." << endl;
79
80   //REDUCE
81   cout << "[Master] Reducing..." << endl;
82   map<string, int> final_result;
83   reduce_function(intermediate_results, final_result);

```

```

81 // OUTPUT
82 ofstream outfile("wordcount_output.txt");
83 for (const auto& pair : final_result) {
84     outfile << pair.first << ":" << pair.second << endl;
85 }
86 outfile.close();
87 cout << "[Master] Success! Unique words: " << final_result.size() <<
88 endl;
89
90 return 0;
}

```

Listing 1: Word Count Implementation (wordcount.cpp)

5 Testing and Results

The testing was conducted on a **Linux Terminal**.

Execution Steps:

- Compilation:** I used `g++` with the `-pthread` flag to link the POSIX threads library required by Linux.

```
g++ -o wordcount wordcount.cpp -std=c++11 -pthread
```

- Execution:** Ran the executable with a sample text file.

```
./wordcount input.txt
```

Results: The terminal output confirmed the successful execution of the MapReduce phases:

- The Master successfully read the file (498 bytes in the sample test).
- 4 parallel threads were launched to process the data chunks.
- The Reduce phase aggregated the results correctly.
- Total unique words found: 69 (matching the content of `input.txt`).

```

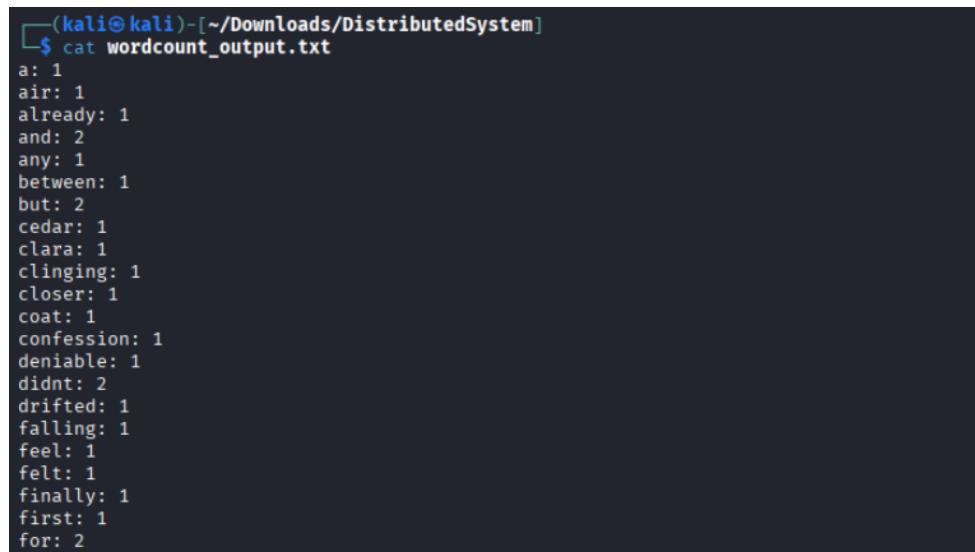
(kali㉿kali)-[~/Downloads/DistributedSystem]
$ ./wordcount input.txt
[Master] Read file size: 498 bytes.
[Master] Starting 4 threads ...
[Master] Map phase complete.
[Master] Reducing ...
[Master] Success! Unique words: 69

```

Figure 2: Terminal Execution Log

Upon successful execution, we also have the result file named `wordcount_output.txt`.

- **Automatic Sorting:** The output is strictly ordered alphabetically. This validates the use of `std::map`, which inherently sorts keys, effectively simulating the **Shuffle & Sort** phase of the MapReduce paradigm without requiring an explicit sorting algorithm.
- **Data Normalization:** Proper nouns such as "Julian" and "Clara" appear as "julian" and "clara". This confirms that the Mapper's text processing logic correctly normalized all tokens to lowercase before counting.
- **Aggregation Accuracy:** High-frequency stop words like "the" (9 occurrences) and "to" (5 occurrences) demonstrate that the Reducer successfully aggregated partial counts from multiple parallel threads into a correct global sum.



```
(kali㉿kali)-[~/Downloads/DistributedSystem]
$ cat wordcount_output.txt
a: 1
air: 1
already: 1
and: 2
any: 1
between: 1
but: 2
cedar: 1
clara: 1
clinging: 1
closer: 1
coat: 1
confession: 1
deniable: 1
didnt: 2
drifted: 1
falling: 1
feel: 1
felt: 1
finally: 1
first: 1
for: 2
```

Figure 3: Output File Content (wordcount_output.txt)