# Practical Work #5 – Longest Path (Mini MapReduce)

Nguyen Tien Dung – 23BI14116

December 4, 2025

## 1 Goal of the practical work

The goal of this practical work is to implement a "Longest Path" application inspired by the MapReduce model. The input is a collection of files, each containing full file paths (for example, the result of running `find /` on different machines). The output is the set of path(s) with the greatest length.

The objectives are:

- Read path lists from one or more input files.
- Apply a **Mapper** to transform each line into a simple key/value representation.
- Apply a **Reducer** to determine the maximum path length and collect all paths with that length.
- Write the longest paths to an output file.
- Follow the logical MapReduce pattern without using an external framework.

## 2 Environment and approach

The implementation is done in C++ on the Kali Linux VM used for previous labs. As with the Word Count lab, I implement a small, single-process "mini MapReduce" engine:

- **Map phase**: read each line (a full path) and emit a pair containing the path length and the path itself.
- **Shuffle / Sort**: collect all length/path pairs and sort them by path length.
- **Reduce phase**: scan the sorted list, find the maximum length, and collect all paths that have that length.

Although the implementation is not distributed, it mirrors the logic of a MapReduce job and is easy to test in a simple lab environment.

## 3 System organization

The Longest Path implementation is contained in a single C++ file:

- `longest_path.cpp`: implements the mini MapReduce engine to find the longest path(s).

The program is called as:

```
./longest_path <output_file> <input_file1> [input_file2 ...]
```

For example:

```
./longest_path longest_output.txt paths1.txt paths2.txt
```

# 4 Mini MapReduce design

## 4.1 Length/path structure

Each line in the input is a candidate path. I represent it as a simple structure:

Listing 1: Struct to hold length and path

```
struct LengthPath {
    int length;
    std::string path;
};
```

The `length` field stores the number of characters in the path (using `std::string::size()`), and `path` stores the full path string.

## 4.2 Mapper: from line to (length, path)

The Mapper operates on a single line of input. For a non-empty line, it computes the path length and emits a `LengthPath` instance.

Listing 2: Mapper function for a single path line

```
LengthPath map_line(const std::string &line) {
    LengthPath lp;
    if (line.empty()) {
        lp.length = -1;
        lp.path = "";
    } else {
        lp.length = static_cast<int>(line.size());
        lp.path = line;
    }
    return lp;
}
```

If the line is empty, the function returns a length of `-1` and an empty path; these entries are ignored later.

## 4.3 Reducer: selecting the longest paths

After the Map phase, I have a vector of `LengthPath` values. To find the longest paths:

1. Remove entries with invalid length (negative).
2. Sort the vector in decreasing order of `length`.
3. Take the first element to get the maximum length.
4. Collect all entries whose `length` equals this maximum.

This logic is implemented in `reduce_all`:

Listing 3: Reducer function to select the longest paths

```
std::vector<LengthPath> reduce_all(std::vector<LengthPath> &intermediate) {
    std::vector<LengthPath> result;
    if (intermediate.empty()) return result;

    // Remove invalid entries
    intermediate.erase(
        std::remove_if(intermediate.begin(), intermediate.end(),
                       [](const LengthPath &lp){ return lp.length < 0; }),
        intermediate.end()
```

```
    );
    if (intermediate.empty()) return result;

    // Sort by length descending, tie-break by path
    std::sort(intermediate.begin(), intermediate.end(),
              [](const LengthPath &a, const LengthPath &b) {
                  if (a.length != b.length) return a.length > b.length;
                  return a.path < b.path;
              });

    int maxLen = intermediate.front().length;
    for (const auto &lp : intermediate) {
        if (lp.length == maxLen) {
            result.push_back(lp);
        } else {
            break; // remaining entries are shorter
        }
    }
    return result;
}
```

# 5 Implementation

## 5.1 Main program

The `main` function reads all input files, applies the Mapper to each line, calls the Reducer, and writes the result:

Listing 4: Main function of longest$_p$*ath.cpp*

```
int main(int argc, char *argv[]) {
    if (argc < 3) {
        std::cerr << "Usage:␣" << argv[0]
                  << "␣<output_file>␣<input_file1>␣[input_file2␣...]\n";
        return 1;
    }

    std::string output_file = argv[1];

    // -------- Map phase --------
    std::vector<LengthPath> intermediate;

    for (int i = 2; i < argc; ++i) {
        std::string input_file = argv[i];
        std::ifstream in(input_file);
        if (!in) {
            std::cerr << "Error:␣cannot␣open␣input␣file:␣"
                      << input_file << "\n";
            continue; // skip files that cannot be opened
        }

        std::string line;
        while (std::getline(in, line)) {
            LengthPath lp = map_line(line);
            if (lp.length >= 0) {
                intermediate.push_back(lp);
            }
        }
```

```
        in.close();
    }

    std::cout << "[MapReduce] Mapped " << intermediate.size()
              << " path entries.\n";

    // -------- Reduce phase --------
    auto result = reduce_all(intermediate);

    if (result.empty()) {
        std::cerr << "[MapReduce] No valid paths found.\n";
        return 1;
    }

    std::cout << "[MapReduce] Longest length = "
              << result.front().length
              << ", number of longest paths = "
              << result.size() << "\n";

    // -------- Write output --------
    std::ofstream out(output_file);
    if (!out) {
        std::cerr << "Error: cannot open output file: "
                  << output_file << "\n";
        return 1;
    }

    for (const auto &lp : result) {
        // Format: length<space>path
        out << lp.length << " " << lp.path << "\n";
    }
    out.close();

    std::cout << "[MapReduce] Result written to: "
              << output_file << "\n";
    return 0;
}
```

# 6 Build commands

The program is compiled with g++:

Listing 5: Compilation command

```
g++ -Wall -O2 longest_path.cpp -o longest_path
```

This creates the executable `longest_path`.

# 7 Execution and test results

## 7.1 Creating test input

To test the Longest Path application, I created two small input files:

```
cat > paths1.txt << 'EOF'
/home/suiikawaii/file.txt
/home/suiikawaii/DistributedSystem/WordCount/input1.txt
```

```
/var/log/syslog
EOF

cat > paths2.txt << 'EOF'
/usr/bin/python3
/home/suiikawaii/very/very/long/path/to/some/important/file.txt
/tmp/x
EOF
```

Each file contains several full paths of different lengths.

### 7.2 Running the program

I then ran the Longest Path program on these two input files:

```
./longest_path longest_output.txt paths1.txt paths2.txt
```

The console output shows how many entries were mapped and information about the longest paths:

```
[MapReduce] Mapped 6 path entries.
[MapReduce] Longest length = 66, number of longest paths = 1
[MapReduce] Result written to: longest_output.txt
```

### 7.3 Inspecting the output

The output file `longest_output.txt` has the format:

```
66 /home/suiikawaii/very/very/long/path/to/some/important/file.txt
```

The number `66` is the length (in characters) of the longest path, and the rest of the line is the full path itself. If there were multiple paths of the same maximum length, they would all appear in the output file.

## 8 Discussion

This practical work demonstrates how the MapReduce pattern can be applied to a different problem than Word Count:

- The **Mapper** converts lines into `(length, path)` pairs instead of `(word, 1)`.
- The **Shuffle/Sort** stage arranges the data so that the maximum length can be easily identified.
- The **Reducer** selects all entries with the global maximum length.

Possible extensions include:

- Splitting the Map phase across multiple threads or processes for very large input datasets.
- Adding filters (for example, ignoring certain directories or file types).
- Computing additional statistics such as average path length, distribution of lengths, etc.

For the scope of this lab, the current implementation is simple, deterministic and clearly illustrates the idea of using MapReduce-style processing for a non-textual aggregation.

## 9 Personal work

For this Longest Path practical work I:

- Designed a MapReduce-style formulation of the Longest Path problem.
- Implemented the Mapper (`map_line`) and Reducer (`reduce_all`) in C++.
- Wrote the main program in `longest_path.cpp` to orchestrate the Map, Reduce and output phases.
- Compiled and tested the program with custom input files.
- Prepared this report documenting the design, implementation steps, build commands and test results.

## 10 Conclusion

In this practical work I implemented a Longest Path application following the MapReduce paradigm. By reusing the same mini MapReduce infrastructure idea as in the Word Count lab, I showed that the model can be applied to different problems such as finding the longest file path in a large set of directory listings. This concludes the series of practical works on distributed and parallel data processing for this course.