Parallel Longest-Path Detection Using Mapper–Reducer Style

Hoang Minh Quan
Student ID: 23BI14371

December 5, 2025

Abstract

This report describes a multithreaded C++ program that finds the longest paths contained in one or more text files. The implementation follows a mapper–reducer style: mapper threads compute the length of each path, while reducer threads determine the maximum length and collect all paths that reach this maximum. The final results are written to an output file together with their length.

Contents

1 Introduction **2**

2 Problem Description **2**

3 System Overview **2**
3.1 High-level Design 2
3.2 Architecture Illustration 2

4 Implementation Details **3**
4.1 Data Structures 3
4.2 Mapper Threads 3
4.3 Reducer Threads 3
4.4 Global Aggregation and Output 4

5 Source Code Listing **4**

6 Usage Example **7**

7 Conclusion **7**

1 Introduction

Searching for the longest element in a large collection is a common pattern in many data-processing tasks. In this assignment we focus on *paths* stored as lines in plain text files. Each line represents a path encoded as a string (for example, a sequence of node identifiers or a filesystem path). Among all available paths, we want to identify those that have the greatest length measured in characters.

To speed up the computation on multi-core machines, the program uses several threads organised as mappers and reducers. The source code is implemented in standard C++ and uses the STL threading facilities. :contentReference[oaicite:1]index=1

2 Problem Description

Given:

- one or more input text files; each line is treated as a candidate path;
- an output file name.

The program must:

1. read all non-empty lines from the input files;
2. compute the length (in characters) of every path;
3. determine the maximum path length over the entire dataset;
4. collect all paths whose length equals this maximum;
5. write the maximum length and each corresponding path to the output file, one per line.

The command-line interface is:

```
longestpath <output_file> <input1> [input2 ...]
```

3 System Overview

3.1 High-level Design

The overall computation is divided into four stages:

1. **Input loading:** all input files are read line by line, and non-empty lines are stored in a single vector of strings.
2. **Map phase:** the vector of paths is split into contiguous chunks. Each mapper thread receives a chunk and produces a list of `(length, path)` pairs.
3. **Reduce phase:** the pairs are partitioned by path length and processed by reducer threads. Each reducer finds the local maximum length among its input pairs and remembers all paths with that length.
4. **Global aggregation:** the maximum length and the corresponding set of paths are computed from all reducers, and the result is written to the output file.

3.2 Architecture Illustration

Figure 1 shows the architecture of the program. Multiple input files are merged into a single pool of paths, which is then processed in parallel by mapper and reducer threads.

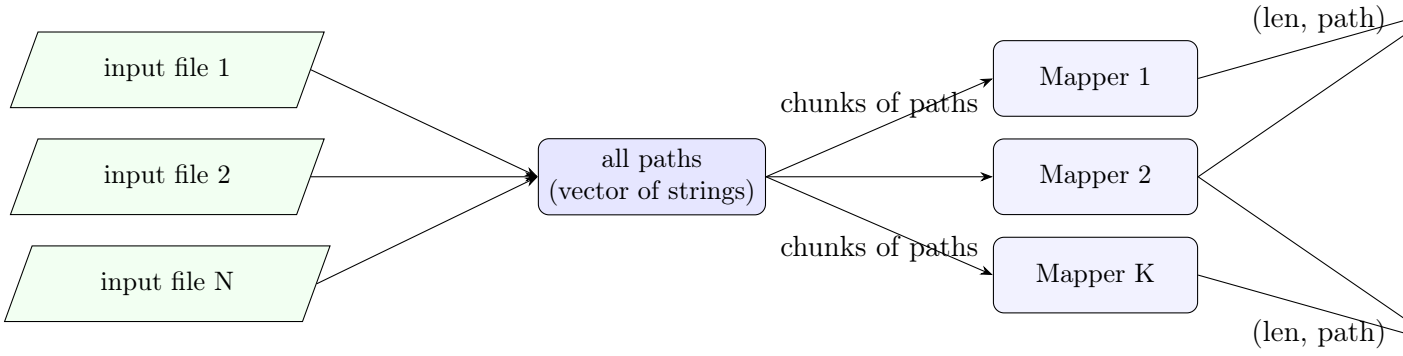

Figure 1: Mapper–reducer style architecture for longest-path detection.

4 Implementation Details

4.1 Data Structures

Two main structures are used in the program:

- a `std::vector<std::string>` storing all paths read from the input files;
- a `ReducerResult` structure for each reducer, holding the local maximum length and the list of paths achieving this length:

```
struct ReducerResult {
    int max_len = 0;
    std::vector<std::string> paths;
};
```

Mapper threads output vectors of `(length, path)` pairs, which are then redistributed among reducers based on the hash of the length.

4.2 Mapper Threads

The function `mapper_worker` receives the global vector of paths and a range `[begin, end)` indicating which paths it should process. For each path, it computes its length (in characters) and appends a pair `(length, path)` to the output vector.

The main thread divides the total number of paths into approximately equal chunks. When the number of paths is smaller than the requested number of mappers, the program automatically reduces the mapper count to avoid idle threads.

4.3 Reducer Threads

Reducer threads take vectors of `(length, path)` pairs as input. Each reducer scans its pairs and keeps track of:

- the largest length it has seen so far, and
- the list of paths that have exactly that length.

When a longer path appears, the reducer updates its local maximum and resets the list of stored paths to contain only the new longest path. When a path ties the current maximum, it is appended to the list.

Hash partitioning on the integer length is used to distribute pairs to reducers:

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

4.4 Global Aggregation and Output

After all reducers have finished, the main thread scans the `ReducerResult` structures to compute a global maximum length and a combined list of paths:

- if a reducer reports a larger maximum, its paths replace the current global list;
- if a reducer reports the same maximum, its paths are appended to the global list.

Finally, each path whose length equals the global maximum is written to the output file in the form:

`<max_length> <path>`

5 Source Code Listing

For completeness, Listing 1 contains the full code of the program.

```
#include <algorithm>
#include <fstream>
#include <iostream>
#include <string>
#include <thread>
#include <unordered_map>
#include <utility>
#include <vector>

struct ReducerResult {
    int max_len = 0;
    std::vector<std::string> paths;
};

void mapper_worker(const std::vector<std::string>& all_paths,
                   size_t begin, size_t end,
                   std::vector<std::pair<int, std::string>>& out_pairs)
                      {
    for (size_t i = begin; i < end; ++i) {
        const std::string& p = all_paths[i];
        int len = static_cast<int>(p.size());
        out_pairs.emplace_back(len, p);
    }
}

void reducer_worker(const std::vector<std::pair<int, std::string>>&
   in_pairs,
                    ReducerResult& out_result) {
    int local_max = 0;
    std::vector<std::string> local_paths;
    for (const auto& kv : in_pairs) {
        int len = kv.first;
        const std::string& path = kv.second;
        if (len > local_max) {
            local_max = len;
            local_paths.clear();
            local_paths.push_back(path);
        } else if (len == local_max) {
            local_paths.push_back(path);
        }
    }
```

```
40      out_result.max_len = local_max;
41      out_result.paths = std::move(local_paths);
42  }
43
44  int main(int argc, char** argv) {
45      if (argc < 3) {
46          std::cerr << "Usage: " << argv[0]
47                    << " <output_file> <input1> [input2 ...] \n";
48          return 1;
49      }
50
51      std::string output_file = argv[1];
52      std::vector<std::string> input_files;
53      for (int i = 2; i < argc; ++i) {
54          input_files.push_back(argv[i]);
55      }
56
57      std::vector<std::string> all_paths;
58      for (const auto& fname : input_files) {
59          std::ifstream in(fname);
60          if (!in.is_open()) {
61              std::cerr << "Cannot open input file: " << fname << "\n";
62              return 1;
63          }
64          std::string line;
65          while (std::getline(in, line)) {
66              if (!line.empty())
67                  all_paths.push_back(line);
68          }
69          in.close();
70      }
71
72      if (all_paths.empty()) {
73          std::cerr << "No paths found in input files.\n";
74          return 1;
75      }
76
77      int num_mappers = 4;
78      int num_reducers = 2;
79      if (static_cast<int>(all_paths.size()) < num_mappers) {
80          num_mappers = static_cast<int>(all_paths.size());
81      }
82      if (num_reducers > num_mappers) {
83          num_reducers = num_mappers;
84      }
85
86      std::vector<std::thread> mapper_threads;
87      std::vector<std::vector<std::pair<int, std::string>>> mapper_outputs
            (num_mappers);
88
89      size_t total = all_paths.size();
90      size_t base_chunk = total / num_mappers;
91      size_t extra = total % num_mappers;
92      size_t start = 0;
93
94      for (int i = 0; i < num_mappers; ++i) {
95          size_t chunk = base_chunk + (i < static_cast<int>(extra) ? 1 :
                0);
```

```
        size_t end = start + chunk;
        mapper_threads.emplace_back(mapper_worker,
                                    std::cref(all_paths),
                                    start, end,
                                    std::ref(mapper_outputs[i]));
        start = end;
    }

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

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

    std::vector<std::thread> reducer_threads;
    std::vector<ReducerResult> reducer_results(num_reducers);

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

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

    int global_max = 0;
    std::vector<std::string> longest_paths;

    for (const auto& res : reducer_results) {
        if (res.max_len > global_max) {
            global_max = res.max_len;
            longest_paths = res.paths;
        } else if (res.max_len == global_max) {
            longest_paths.insert(longest_paths.end(),
                                 res.paths.begin(), res.paths.end());
        }
    }

    if (global_max == 0) {
        std::cerr << "Could not compute longest path length.\n";
        return 1;
    }

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

    for (const auto& p : longest_paths) {
        if (static_cast<int>(p.size()) == global_max)
```

```
            out << global_max << " " << p << "\n";
        }
        out.close();

        std::cout << "Longest path length: " << global_max
                  << ", number of paths: " << longest_paths.size() << "\n";
        std::cout << "Results written to: " << output_file << "\n";

        return 0;
}
```

Listing 1: C++ implementation of parallel longest-path detection

6 Usage Example

After compiling the program with a C++17-capable compiler, for example:

```
g++ -std=c++17 -pthread longestpath.cpp -o longestpath
```

it can be executed as:

```
./longestpath results.txt paths1.txt paths2.txt
```

The file `results.txt` will then contain the maximum path length followed by each path that achieves this length.

7 Conclusion

The longest-path program illustrates how a mapper–reducer style design can be used in a multithreaded C++ application. By splitting the input set of paths among mapper threads and combining partial maxima with reducers, the solution is able to exploit several CPU cores while keeping the code relatively simple. The same pattern could be reused for other aggregation tasks, such as finding the shortest path, the most frequent pattern, or computing summary statistics over large datasets.