Word Count Using Custom MapReduce Implementation

Nguyen Thanh Nam - 22BI13325

Introduction

This word count program uses a custom MapReduce framework in C++. Processing large text files by dividing the workload into smaller parts handled in parallel using multithreading. This ensures faster processing while maintaining accuracy. The output includes the total number of words and their respective frequencies, sorted in descending order for easy analysis.

Why?

Customized implementation of MapReduce was created because there are no native MapReduce libraries available for C++. The reason why C++ was chosen as the programming language is its high efficiency and precise memory control, which are essential when dealing with large datasets. Additionally, multithreading allows different parts of the text to be processed simultaneously, significantly reducing computation time and making the program highly performant.

Mapper and Reducer Design

The MapReduce process was divided into two main stages:

Mapper

The mapper function processes a fragment of text to count the frequency of each word. Performs the following steps:

1. Splits the text into words, removing punctuation and converting them to lowercase.
2. Count the occurrence of each word in the chunk.
3. Outputs a local word count map for the chunk.

Reducer

The reducer function aggregates the local word count maps into a global word count map. It:

1. Locks access to the global map using a mutex for thread safety.
2. Adds the counts from the local map to the global map.

Visualization of MapReduce Process

```
Input Text:  "Word count example for MapReduce framework"

[Mapper Phase]
Chunk 1: "Word count example"
Chunk 2: "For MapReduce framework"

Mapper Output:
Chunk 1 -> {"word": 1, "count": 1, "example": 1}
Chunk 2 -> {"for": 1, "mapreduce": 1, "framework": 1}

[Reducer Phase]
Global Word Count:
{"word": 1, "count": 1, "example": 1, "for": 1, "mapreduce": 1, "framework": 1}

Sorted Output:
framework: 1
mapreduce: 1
example: 1
count: 1
word: 1
for: 1
```

Responsibilities

The implementation was designed with distinct responsibilities:

- **Main Thread:** Reads the input text, splits it into fragments, and manages the execution mapper and reducer threads.
- **Mapper Threads:** Process chunks of text and produce local word count maps.
- **Reducer Threads:** Aggregate local maps into the global word count map.

Conclusion

The word count program represented how a customized MapReduce framework can be designed and implemented to handle large-scale data processing. By using C++ and multithreading, the solution is both efficient and scalable. This project highlights how tasks like word counting, which involve repetitive and parallelizable operations, can be optimized to run quickly and effectively, even on massive datasets.