Project Report

Session Fall 2024 – BSAI

Submitted to:
Ma’am Kanza Hamid
Ma’am Palwasha Zahid

Submitted by:
Hasnain Ibrar 22i-0530

Muhammad Ali Taha 22i-0529

Ali Hussain 22i-1388-0512

Department of Artificial Intelligence

National University of Computer and Emerging sciences,

FAST

Report on Multithreaded MapReduce Implementation in C++

Introduction

This report describes a multithreaded implementation of the MapReduce framework using C++. The program processes an input string by dividing it into chunks, performing a parallel map operation to generate key-value pairs, grouping keys during the shuffle phase, and finally reducing the data to count occurrences of each key. The program is designed to demonstrate concepts of parallel processing, thread safety, and efficient data aggregation.

Program Overview

The program is divided into several key phases:

1. **Input Handling**: Accepts a string input and a specified number of chunks.
2. **Chunk Splitting**: Splits the input string into equal-sized parts (chunks) for processing.
3. **Map Phase**: Processes each chunk in parallel to generate key-value pairs.
4. **Shuffle Phase**: Groups keys and counts their occurrences.
5. **Reduce Phase**: Consolidates key-value pairs to produce final counts.

The program uses **POSIX threads (pthreads)** for multithreading and employs mutex locks to ensure thread safety during shared data access.

Detailed Workflow

1. Input Handling

The program accepts two inputs from the user:

- A string to be processed.
- The number of chunks to divide the input into.

If the input string is empty, the program skips computation, as demonstrated in the test cases.

2. Chunk Splitting

Function: `split_string_into_chunks(const char *input, int no_of_chunks)`

- **Purpose**: Divides the input string into approximately equal-sized chunks.
- **Implementation**:
    - Counts the total number of words in the input string.
    - Calculates the number of words per chunk and distributes remaining words evenly.
    - Assigns words to chunks in a round-robin fashion.

Example Output:
Chunk 1: apple banana
Chunk 2: orange grape
Chunk 3: pineapple

3. Map Phase

Function: `mapphase(void *arg)`

- **Purpose**: Processes each chunk in parallel to produce key-value pairs `(word, 1)`.

- **Implementation**:

    - Each chunk is processed by a separate thread.
    - Words in the chunk are tokenized and added to `map_output` with a count of 1.
    - Thread-safe access to shared resources is ensured using `pthread_mutex_lock()` and `pthread_mutex_unlock()`.

Example Output:
(apple,1)
(banana,1)
(orange,1)
(grape,1)
(pineapple,1)

4. Shuffle Phase

Function: `shufflephase()`

- **Purpose**: Groups identical keys and tracks their occurrences.
- **Implementation**:
    - Iterates over `map_output` to identify unique keys.
    - Keys are stored in `shuffled_keys`, and their occurrences are tracked in `shuffled_occurences`.
    - Unique keys are added to the list only if they are not already present.

Example Output:

apple: [1]
banana: [1]
orange: [1]
grape: [1]
pineapple: [1]

5. Reduce Phase

Function: `reducephase(void *arg)`

- **Purpose**: Computes the total count for each unique key.
- **Implementation**:
    - Each key is processed by a separate thread.
    - The thread counts all occurrences of the key in `shuffled_occurences` and prints the result.
    - Mutex locking ensures thread-safe counting.

Example Output:

Key: apple Count: 1
Key: banana Count: 1
Key: orange Count 1
Key: grape Count 1
Key: pineapple Count 1

6. Test Cases

Function: `test_mapreduce(string input, int no_of_chunks)`

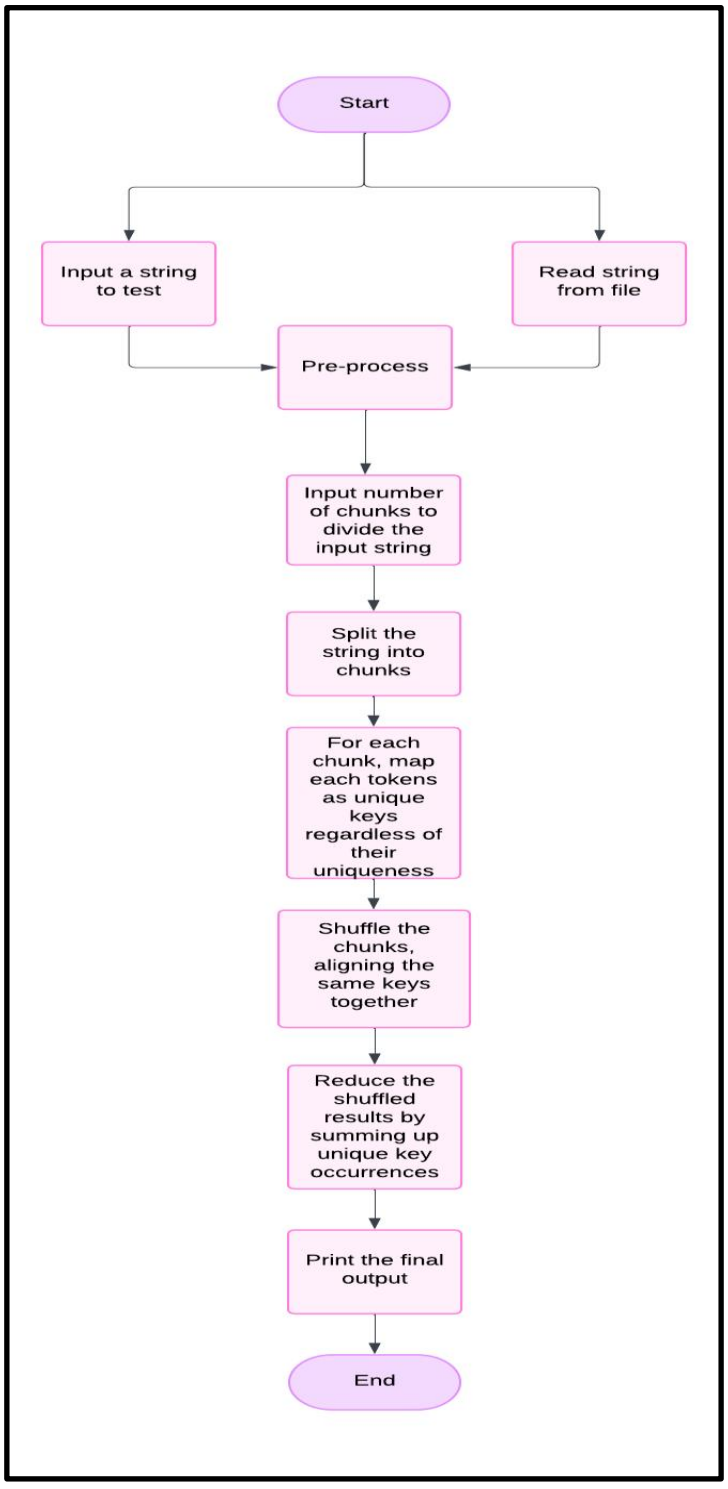
The program includes a function for running predefined test cases to validate correctness under different scenarios, including:

- Mixed case input.
- Empty strings.
- Special characters.

Key Features

1. *Multithreading:*
    - The program uses POSIX threads to parallelize the map and reduce phases, improving performance for large datasets.
    - Each thread independently processes a chunk (map phase) or a unique key (reduce phase).
2. *Thread Safety:*
    - Mutex locks ensure that shared resources, such as `map_output` and `shuffled_occurences`, are accessed safely.
3. *Dynamic Input Handling:*
    - Supports variable input sizes and chunk counts, allowing flexibility in processing.
4. *Test Case Functionality:*
    - The `test_mapreduce` function validates the program against edge cases, such as empty inputs or repeated words.

Flow Chart


Start
Input a string to test
Read string from file
Pre-process
Input number of chunks to divide the input string
Split the string into chunks
For each chunk, map each tokens as unique keys regardless of their uniqueness
Shuffle the chunks, aligning the same keys together
Reduce the shuffled results by summing up unique key occurrences
Print the final output
End

Sample Test Case Outputs

Test Case 1: *Regular Input*

Input: `"apple banana orange apple banana orange , 2"`

```
Test Case: Input = "apple banana orange apple banana orange", Chunks = 2
Chunks after splitting:
Chunk 1: apple banana orange
Chunk 2: apple banana orange

Mapping Phase:
(apple,1)
(banana,1)
(orange,1)
(apple,1)
(banana,1)
(orange,1)

Shuffle Phase:
apple: [1, 1]
banana: [1, 1]
orange: [1, 1]

Key: orange Count: 2
Key: banana Count: 2
Key: apple Count: 2
```

Test Case 2: *Case Sensitivity*

Input: `"Apple APPLE aPpLe"`

```
Executing Test Cases...
Test Case: Input = "Apple apple APPLE aPPle", Chunks = 2
Chunks after splitting:
Chunk 1: apple apple
Chunk 2: apple apple

Mapping Phase:
(apple,1)
(apple,1)
(apple,1)
(apple,1)

Shuffle Phase:
apple: [1, 1, 1, 1]

Key: apple Count: 4
```

Test Case 3: *Empty Input*

Test Case: Input is an empty string. No computation performed.

```
Executing Test Cases...
Test Case: Input is an empty string. No computation performed.
```

Test Case 4: *Special Characters / Numbers*

```
Executing Test Cases...
Test Case: Input = "123 1 123 1 abc abc abd @#$$ ABC APple apple", Chunks = 3
Chunks after splitting:
Chunk 1: 123 1 123 1
Chunk 2: abc abc abd @#$$
Chunk 3: abc apple apple

Mapping Phase:
(123,1)
(1,1)
(123,1)
(1,1)
(abc,1)
(abc,1)
(abd,1)
(@#$$,1)
(abc,1)
(apple,1)
(apple,1)

Shuffle Phase:
123: [1, 1]
1: [1, 1]
abc: [1, 1, 1]
abd: [1]
@#$$: [1]
apple: [1, 1]

Key: 1 Count: 2
Key: abc Count: 3
Key: abd Count: 1
Key: @#$$ Count: 1
Key: apple Count: 2
Key: 123 Count: 2
```

Advantages of the Program

1. *Scalability:*
    - Efficiently handles large datasets by parallelizing the map and reduce phases.
2. *Simplicity:*
    - Easy-to-follow implementation suitable for learning concepts of MapReduce.
3. *Customizability:*
    - Flexible input and chunk handling make the program adaptable for different data sizes and structures.

Potential Improvements

1. *Dynamic Data Structures:*
    - Replace fixed-size arrays with dynamic data structures (e.g., `std::vector`) to eliminate size constraints.
2. *Enhanced Performance:*
    - Optimize shuffle and reduce phases with more efficient algorithms or data structures like hash maps.
3. *Integration:*
    - Extend functionality to process data from external files, such as JSON or CSV formats.

Conclusion

This multithreaded MapReduce implementation demonstrates a foundational approach to distributed data processing using C++. It efficiently divides tasks into independent operations, leverages multithreading for parallelism, and ensures correctness through mutex locking. This program provides an excellent learning resource for understanding the principles of MapReduce and parallel programming.