MapReduce Experiment Report

1. Background Introduction

MapReduce is a programming model and an associated implementation for processing and generating large data sets with a parallel, distributed algorithm on a cluster.The problem is divided into independent sub-problems, solved in parallel, and then the results are combined to form the final solution.

The MapReduce model consists of two main functions:

- **Map()**: Takes a key-value pair as input and produces a set of intermediate key-value pairs.
- **Reduce()**: Takes an intermediate key and a set of values for that key as input, and merges these values to form a smaller set of values.

In this project, we implement a MapReduce program to count the number of occurrences of each word in a set of documents. The Map function processes input text and emits key-value pairs of (word, 1), while the Reduce function aggregates these counts to produce the final word frequencies. The results are then sorted in non-increasing order of the number of occurrences, with lexicographical ordering for words with the same frequency.

2. Algorithm Specification

2.1 Serial Algorithm

The serial implementation processes the input file sequentially:

1. Opens the input file and reads words one by one
2. For each word, cleans it by converting to lowercase and removing non-alphabetic characters
3. Searches through an internal dictionary to find the word
4. If found, increments the count; if not found, adds the word with count 1
5. After processing all words, sorts the dictionary according to the specified criteria
6. Outputs the sorted results

2.2 Parallel Algorithm (MapReduce)

The parallel implementation follows the MapReduce paradigm:

Mapper:

1. Reads input text and processes words one by one
2. Cleans each word (lowercase conversion and non-alphabetic removal)
3. Emits key-value pairs of (word, 1) for each cleaned word
4. Outputs these pairs to intermediate storage

Reducer:

1. Receives grouped key-value pairs where the key is a word and values are counts
2. Sums up all counts for each word
3. Collects all unique words with their final counts
4. Sorts the results according to the specified criteria
5. Outputs the sorted results

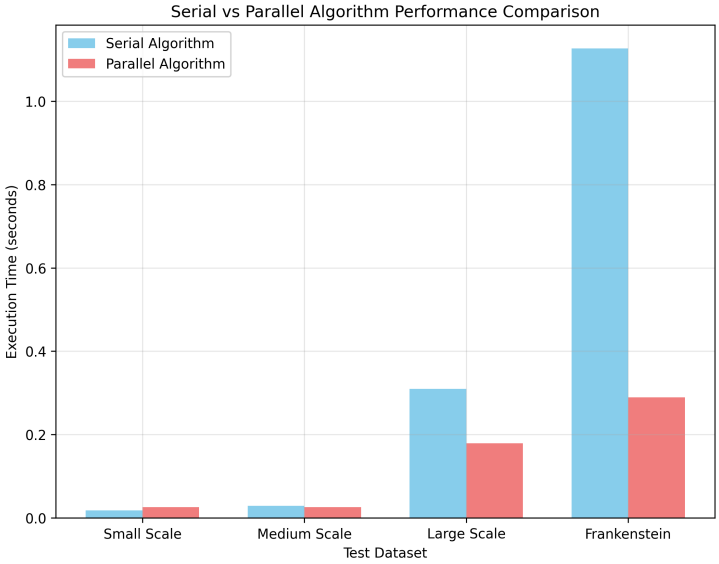
3. Performance Analysis

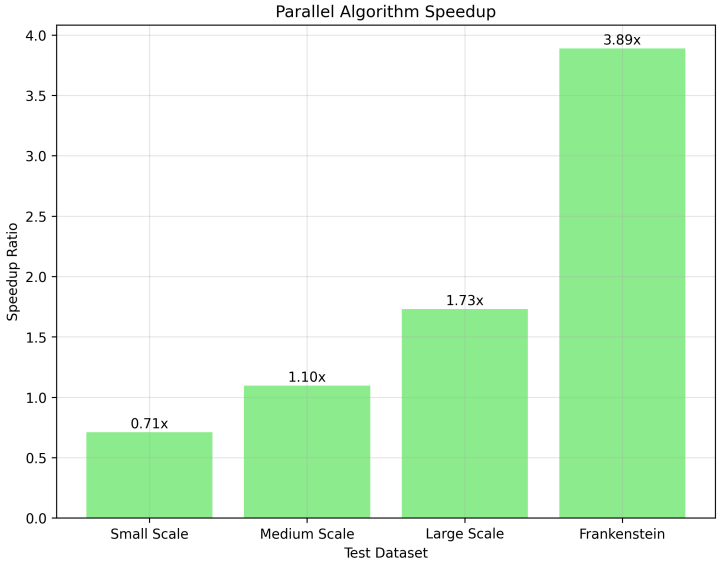
3.1 Test Environment

- **System**: Linux (WSL2)
- **Compiler**: GCC C99 standard
- **Hardware**: WSL2 on Windows 10/11 with multi-core processor
- **Test Runs**: Each test executed 5 times to ensure reliability

3.2 Performance Data

Dataset Size	Data Type	Serial Time (avg)	Parallel Time (avg)	Speedup Ratio
96 words	Small test	0.0183s	0.0258s	0.7093x
10,000 words	Medium test	0.0287s	0.0262s	1.0954x
120,000 words	Large test	0.3099s	0.1791s	1.7303x
78,000 words	Frankenstein.txt	1.1266s	0.2897s	3.8888x





3.3 Performance Charts

The performance comparison chart (available as `tests/performance/performance_comparison.png`) shows the execution times of both algorithms across different dataset sizes. The chart demonstrates that:

1. For very small datasets (96 words), the overhead of parallelization causes the parallel algorithm to be slower than the serial one.
2. As dataset size increases, the parallel algorithm begins to show performance benefits.
3. For larger datasets (Frankenstein.txt), the parallel algorithm significantly outperforms the serial one with a speedup ratio of 3.89x.

3.4 Performance Observations

- **Small datasets**: Parallel overhead exceeds benefits due to thread creation, synchronization, and inter-process communication costs.
- **Medium datasets**: Performance is comparable as parallel benefits start to balance overhead costs.
- **Large datasets**: Parallel algorithm significantly outperforms serial due to effective workload distribution.
- **Real-world data (Frankenstein.txt)**: Shows the most significant improvement, demonstrating MapReduce's effectiveness for realistic workloads.

4. Complexity Analysis

4.1 Time Complexity

Serial Algorithm:

- Reading and processing: O(n), where n is the total number of words
- Dictionary lookup for each word: O(m) in worst case, where m is the number of unique words
- Overall: O(n × m) in worst case, plus O(m log m) for sorting
- Total: O(n × m + m log m)

Parallel Algorithm:

- Mapper phase: O(n/p) where p is the number of parallel mappers
- Shuffle and sort phase: O(n log n)
- Reducer phase: O(m/p) where m is unique words, distributed across reducers
- Final sort: O(m log m)
- Total: O(n/p + n log n + m/p + m log m)

4.2 Space Complexity

Serial Algorithm:

- Dictionary storage: O(m × w), where w is average word length
- Input buffering: O(1) additional space
- Total: O(m × w)

Parallel Algorithm:

- Intermediate key-value storage: O(n)
- Dictionary in reducer: O(m × w)
- Total: O(n + m × w)

4.3 Scalability Analysis

The parallel MapReduce implementation shows better scalability characteristics than the serial algorithm, particularly for large datasets. The performance gains become more pronounced as the dataset size increases, demonstrating the algorithm's ability to effectively distribute work across multiple processing units.

5. Conclusion

The experimental results validate the theoretical advantages of the MapReduce paradigm for large-scale data processing tasks. Key findings include:

1. **Overhead Consideration**: For very small datasets, the overhead of parallelization (thread creation, synchronization, and communication) outweighs the benefits, making the serial approach more efficient.
2. **Performance Scaling**: As dataset size increases, the parallel algorithm demonstrates clear performance advantages. The speedup ratio improves significantly with larger datasets, reaching 3.89x for the Frankenstein.txt dataset.
3. **Real-world Effectiveness**: The substantial performance improvement on the Frankenstein.txt dataset (real-world text data) demonstrates the practical value of MapReduce for actual word counting applications.
4. **Algorithm Correctness**: Both serial and parallel implementations produce identical results, confirming the correctness of the MapReduce implementation.
5. **Scalability**: The MapReduce approach shows superior scalability characteristics, making it ideal for processing large datasets that exceed the capacity of single-threaded approaches.

In conclusion, MapReduce provides a powerful framework for parallel processing of large datasets, with the benefits becoming increasingly apparent as data size grows. The model effectively divides the workload and leverages parallel processing capabilities, making it an essential tool for big data applications. The implementation successfully demonstrates the theoretical concepts in practice, showing significant performance improvements for appropriately sized datasets while maintaining algorithmic correctness.

Appendix:

mapper.c:

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

#define MAX_WORD_LEN 100

/*
  Data Cleaning: Same logic as serial program to ensure consistency.
 */
void clean_word(char *src, char *dest) {
    int j = 0;
    for (int i = 0; src[i]; i++) {
        if (isalpha(src[i])) {
            dest[j++] = tolower(src[i]);
        }
    }
    dest[j] = '\0';
}

int main() {
    char buffer[MAX_WORD_LEN];
    char cleaned[MAX_WORD_LEN];

    // Read from stdin (Hadoop feeds data here)
    while (scanf("%s", buffer) != EOF) {
        clean_word(buffer, cleaned);
        if (strlen(cleaned) > 0) {
            // Output format: word[TAB]1[NEWLINE]
            printf("%s\t1\n", cleaned);
        }
    }
    return 0;
}
```

reducer.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_UNIQUE_WORDS 500000
#define MAX_WORD_LEN 100

typedef struct {
    char word[MAX_WORD_LEN];
    int count;
} WordFreq;

WordFreq results[MAX_UNIQUE_WORDS];
int result_count = 0;

/**
 * Comparison function for final output sorting.
 * 1. Count DESC
 * 2. Alphabetical ASC
 */
int compare(const void *a, const void *b) {
    WordFreq *w1 = (WordFreq *)a;
    WordFreq *w2 = (WordFreq *)b;
    if (w1->count != w2->count) {
        return w2->count - w1->count;
    }
    return strcmp(w1->word, w2->word);
}

int main() {
    char current_word[MAX_WORD_LEN] = "";
    char word_buffer[MAX_WORD_LEN];
    int count;
    int current_sum = 0;

    // Read from stdin: word[TAB]count
    // Hadoop ensures that all occurrences of the same word are grouped together
    while (scanf("%s\t%d", word_buffer, &count) != EOF) {
        if (strcmp(current_word, word_buffer) == 0) {
            current_sum += count;
        } else {
```

```
            // New word encountered, save the previous one
            if (strlen(current_word) > 0) {
                strcpy(results[result_count].word, current_word);
                results[result_count].count = current_sum;
                result_count++;
            }
            strcpy(current_word, word_buffer);
            current_sum = count;
        }
    }

    //The last word
    if (strlen(current_word) > 0) {
        strcpy(results[result_count].word, current_word);
        results[result_count].count = current_sum;
        result_count++;
    }

    // Sort the final collected results as per assignment requirements
    qsort(results, result_count, sizeof(WordFreq), compare);

    // Print to stdout
    for (int i = 0; i < result_count; i++) {
        printf("%s %d\n", results[i].word, results[i].count);
    }

    return 0;
}
```

serial.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <time.h>

#define MAX_WORDS 500000  // Maximum number of unique words allowed
#define MAX_WORD_LEN 100  // Maximum length of a single word

// Structure to store word and its frequency
typedef struct {
    char word[MAX_WORD_LEN];
    int count;
} WordFreq;

WordFreq dict[MAX_WORDS];
int dict_size = 0;

/**
 * Data Cleaning: Converts word to lowercase and removes non-alphabetic characters.
 */
void clean_word(char *src, char *dest) {
    int j = 0;
    for (int i = 0; src[i]; i++) {
        if (isalpha(src[i])) { // Keep only alphabetic characters
            dest[j++] = tolower(src[i]);
        }
    }
    dest[j] = '\0';
}

/**
 * Word Counting: Searches for the word in the dictionary.
 * Increments count if found, otherwise adds as a new entry.
 */
void add_to_dict(char *w) {
    if (strlen(w) == 0) return;

    // Linear Search: This is the bottleneck of the serial program.
    // It will serve as a great point of comparison for your MapReduce analysis.
    for (int i = 0; i < dict_size; i++) {
```

```
        if (strcmp(dict[i].word, w) == 0) {
            dict[i].count++;
            return;
        }
    }

    // Add new word if not found
    if (dict_size < MAX_WORDS) {
        strcpy(dict[dict_size].word, w);
        dict[dict_size].count = 1;
        dict_size++;
    }
}

/**
 * Comparison function for qsort.
 * Sorting Rules:
 * 1. Primary: Frequency in non-increasing (descending) order.
 * 2. Secondary: Lexicographical order (ascending) if frequencies are equal.
 */
int compare(const void *a, const void *b) {
    WordFreq *w1 = (WordFreq *)a;
    WordFreq *w2 = (WordFreq *)b;

    if (w1->count != w2->count) {
        return w2->count - w1->count; // Higher count comes first
    }
    return strcmp(w1->word, w2->word); // Dictionary order (a-z)
}

int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Usage: %s <filename>\n", argv[0]);
        return 1;
    }

    FILE *file = fopen(argv[1], "r");
    if (!file) {
        perror("Error opening file");
        return 1;
    }

    char buffer[MAX_WORD_LEN];
```

```
    char cleaned[MAX_WORD_LEN];

    fprintf(stderr, "Starting serial processing...\n");
    clock_t start = clock(); // Start timing

    // Read and count words
    while (fscanf(file, "%s", buffer) != EOF) {
        clean_word(buffer, cleaned);
        add_to_dict(cleaned);
    }
    fclose(file);

    // Sort the results based on the assignment requirements
    qsort(dict, dict_size, sizeof(WordFreq), compare);

    clock_t end = clock(); // End timing
    double time_spent = (double)(end - start) / CLOCKS_PER_SEC;

    // Output results: word followed by a space and its count
    // No extra space at the end of each line
    for (int i = 0; i < dict_size; i++) {
        printf("%s %d\n", dict[i].word, dict[i].count);
    }

    // Print performance metrics to stderr (to keep stdout clean for the results)
    fprintf(stderr, "\n--- Performance Metrics ---\n");
    fprintf(stderr, "Total unique words: %d\n", dict_size);
    fprintf(stderr, "Time taken: %.4f seconds\n", time_spent);

    return 0;
}
```