**19CSE313 Parallel File Processing Case Study Report**

- GROUP -19

|  |  |
| --- | --- |
| Roll No. | Name |
| CB.EN.U4CSE22402 | Poornima A |
| CB.EN.U4CSE22423 | Sharmila M |
| CB.EN.U4CSE22429 | Polisetty Ramya Sri |

**1. Introduction**

The ParallelWordCounter program exemplifies an efficient approach to processing multiple text files concurrently, leveraging Java's threading capabilities to maximize computational resources. This report analyzes a Java implementation designed to count words, characters, and lines across multiple text files in parallel while also tracking word frequency.

Traditional sequential file processing approaches can be inefficient, particularly when dealing with large volumes of data or numerous files. The parallel processing model presented in this case study addresses these limitations by utilizing multi-threading to distribute the workload across available CPU cores, significantly improving processing throughput and efficiency.

The solution implements a sophisticated two-level parallelism approach:

1. File-level parallelism: Processing multiple files simultaneously
2. Content-level parallelism: Analyzing different chunks of the same file concurrently

This design maximizes CPU utilization while maintaining an organized approach to resource management and result aggregation.

**2. System Architecture**

The ParallelWordCounter implements a structured architecture with the following key components:

**2.1 Core Components**

1. **Main Controller (ParallelWordCounter class)**
   * Orchestrates the overall processing flow
   * Manages thread pools and batch processing
   * Handles result aggregation and reporting
2. **Statistics Container (TextStats class)**
   * Encapsulates analysis metrics for files, batches, and overall results
   * Provides thread-safe methods for updating and merging statistics
   * Stores word frequencies, line counts, character counts, and word counts
3. **Thread Pools**
   * File Thread Pool: Dedicated to file-level parallelism
   * Content Thread Pool: Focused on chunk-level parallelism within files
   * Both sized according to available system processors
4. **Processing Units**
   * Files: Individual text documents to analyze
   * Batches: Groups of files processed together (configurable size)
   * Chunks: Segments of file content processed in parallel

**2.2 System Configuration**

The system incorporates several configurable parameters:

* NUM\_THREADS: Automatically set to the number of available processors
* BATCH\_SIZE: Number of files processed in each batch (default: 5)
* Chunk size: Number of lines processed together within a file (fixed at 100)

These parameters allow the system to adapt to different hardware configurations and workload characteristics.

**3. Processing Workflow**

The file processing workflow follows a structured approach with multiple levels of parallelism:

**3.1 Initialization Phase**

1. **Command-line Argument Validation**
   * The program validates that a directory path is provided
   * Confirms the path exists and is a directory
2. **File Discovery**
   * Uses Java NIO to list all regular files in the specified directory
   * Collects paths into a list for batch processing
3. **Batch Planning**
   * Calculates the number of batches based on file count and batch size
   * Creates a container for final statistics aggregation

**3.2 Batch Processing**

For each batch of files:

1. **Batch Initialization**
   * Extracts a subset of files based on batch number and size
   * Creates a statistics container for the current batch.
2. **Parallel File Processing**
   * Submits each file for processing to the file thread pool
   * Each file is processed by a separate thread
   * Tracks which thread handles each file
3. **Result Collection**
   * Collects and merges file-level statistics
   * Displays thread assignments and file summaries
   * Updates batch statistics

**3.3 File Processing**

Each file is processed as follows:

1. **File Reading**
   * Opens the file using a BufferedReader with UTF-8 encoding
   * Reads the file line by line
2. **Chunk Creation**
   * Accumulates lines into chunks (100 lines each)
   * Creates a copy of each chunk for parallel processing
3. **Parallel Chunk Analysis**
   * Submits each chunk to the content thread pool
   * Multiple chunks from the same file are analyzed concurrently
4. **Synchronization**
   * Waits for all chunk processing to complete
   * Returns aggregated file statistics

**3.4 Text Analysis**

Each chunk of text is analyzed as follows:

1. **Pattern Matching**
   * Uses regex pattern to identify words (letters, numbers, apostrophes)
   * Processes text in lowercase for consistent word counting
2. **Local Counting**
   * Counts lines, characters, and words
   * Accumulates word frequencies in a local map
3. **Thread-Safe Merging**
   * Merges local word counts into the file's global statistics
   * Uses concurrent collections and synchronized methods

**3.5 Results Aggregation**

1. **File-to-Batch Merging**
   * Combines statistics from all files in a batch
   * Displays summary statistics for the batch
2. **Batch-to-Final Merging**
   * Aggregates batch statistics into final results
   * Produces comprehensive summary across all files
3. **Resource Cleanup**
   * Shuts down thread pools after processing completes

**4. Synchronization Mechanisms**

The program employs several synchronization techniques to ensure thread safety and data consistency:

**4.1 Thread-Safe Collections**

1. **ConcurrentHashMap**
   * Used for word frequency counting
   * Provides thread-safe operations without explicit locking
   * Enables efficient concurrent access for updates
2. **Thread Assignment Tracking**
   * Uses ConcurrentHashMap to record which thread processes each file
   * Ensures visibility of assignments across threads

**4.2 Method Synchronization**

1. **Synchronized Methods**
   * The merge() method is synchronized to prevent concurrent modification
   * Ensures atomic updates when combining statistics
2. **Future Objects**
   * Uses Future objects to track task completion
   * Waits for all parallel tasks to finish before proceeding

**4.3 Output Synchronization**

1. **System.out Lock**
   * Synchronizes on System.out when displaying results
   * Prevents interleaved output from multiple threads
2. **Collected Display**
   * Gathers all results before displaying
   * Organizes output in a hierarchical manner

**4.4 Local-to-Global Aggregation**

1. **Local Counting First**
   * Performs counting in thread-local variables
   * Reduces contention on shared data structures
2. **Batch-wise Processing**
   * Processes files in batches sequentially
   * Allows parallelism within batches while maintaining order

**5. Code and Output**

import java.io.\*;

import java.nio.charset.StandardCharsets;

import java.nio.file.\*;

import java.util.\*;

import java.util.concurrent.\*;

import java.util.regex.\*;

import java.util.stream.Collectors;

public class ParallelWordCounter {

    private static final int NUM\_THREADS = Runtime.getRuntime().availableProcessors();

    private static final int BATCH\_SIZE = 5;

    private static final ExecutorService fileThreadPool = Executors.newFixedThreadPool(NUM\_THREADS);

    private static final ExecutorService contentThreadPool = Executors.newFixedThreadPool(NUM\_THREADS);

    static class TextStats {

        String name;

        String threadName;

        ConcurrentHashMap<String, Integer> wordCounts = new ConcurrentHashMap<>();

        int lineCount = 0;

        int charCount = 0;

        int wordCount = 0;

        List<String> fileContents = new ArrayList<>();

        public TextStats(String name, String threadName) {

            this.name = name;

            this.threadName = threadName;

        }

        public synchronized void merge(TextStats other) {

            mergeWordCounts(this.wordCounts, other.wordCounts);

            this.lineCount += other.lineCount;

            this.charCount += other.charCount;

            this.wordCount += other.wordCount;

            this.fileContents.addAll(other.fileContents);

        }

    }

    public static void main(String[] args) throws IOException, InterruptedException, ExecutionException {

        if (args.length < 1) {

            System.out.println("Usage: java ParallelTextAnalyzer <directory\_path>");

            return;

        }

        Path folderPath = Paths.get(args[0]);

        if (!Files.exists(folderPath) || !Files.isDirectory(folderPath)) {

            System.out.println("Error: Directory does not exist: " + folderPath);

            return;

        }

        List<Path> files = Files.list(folderPath)

                .filter(Files::isRegularFile)

                .collect(Collectors.toList());

        if (files.isEmpty()) {

            System.out.println("No files found in the directory: " + folderPath);

            return;

        }

        int batchCount = (int) Math.ceil((double) files.size() / BATCH\_SIZE);

        System.out.println("\n=====================================");

        System.out.println("Total files: " + files.size());

        System.out.println("Processing in " + batchCount + " batch(es) in order...");

        System.out.println("=====================================\n");

        TextStats finalStats = new TextStats("All Files", "N/A");

        for (int batchNum = 0; batchNum < batchCount; batchNum++) {

            TextStats batchStats = processBatch(files, batchNum);

            finalStats.merge(batchStats);

            synchronized (System.out) {

                System.out.println("\n=====================================");

                displayStats(batchStats, "BATCH " + (batchNum + 1) + " SUMMARY");

            }

        }

        System.out.println("\n=====================================");

        displayStats(finalStats, "FINAL RESULTS (ALL FILES)");

        fileThreadPool.shutdown();

        contentThreadPool.shutdown();

    }

    private static TextStats processBatch(List<Path> allFiles, int batchNum) throws InterruptedException, ExecutionException {

        int start = batchNum \* BATCH\_SIZE;

        int end = Math.min(start + BATCH\_SIZE, allFiles.size());

        List<Path> batchFiles = allFiles.subList(start, end);

        synchronized (System.out) {

            System.out.println("\n-------------------------------------");

            System.out.println("BATCH " + (batchNum + 1) + " STARTING...");

            System.out.println("Processing files " + (start + 1) + " to " + end + " of " + allFiles.size());

            System.out.println("-------------------------------------");

        }

        List<Future<TextStats>> fileFutures = new ArrayList<>();

        ConcurrentHashMap<String, String> threadAssignments = new ConcurrentHashMap<>();

        for (Path file : batchFiles) {

            fileFutures.add(fileThreadPool.submit(() -> {

                TextStats stats = processFile(file);

                threadAssignments.put(stats.name, stats.threadName);

                return stats;

            }));

        }

        TextStats batchStats = new TextStats("Batch " + (batchNum + 1), "N/A");

        // Collect thread assignments before printing

        List<TextStats> fileStatsList = new ArrayList<>();

        for (Future<TextStats> future : fileFutures) {

            fileStatsList.add(future.get());

        }

        synchronized (System.out) {

            System.out.println("\nTHREAD ASSIGNMENTS (Batch " + (batchNum + 1) + "):");

            threadAssignments.forEach((fileName, threadName) ->

                System.out.printf("Thread %-20s -> File: %s%n", threadName, fileName)

            );

            for (TextStats fileStats : fileStatsList) {

                batchStats.merge(fileStats);

                displayStats(fileStats, "FILE SUMMARY: " + fileStats.name);

            }

        }

        return batchStats;

    }

    private static TextStats processFile(Path file) throws IOException, InterruptedException, ExecutionException {

        String fileName = file.getFileName().toString();

        String threadName = Thread.currentThread().getName();

        TextStats fileStats = new TextStats(fileName, threadName);

        List<Future<Void>> chunkFutures = new ArrayList<>();

        try (BufferedReader reader = Files.newBufferedReader(file, StandardCharsets.UTF\_8)) {

            List<String> chunk = new ArrayList<>();

            String line;

            int chunkSize = 100;

            while ((line = reader.readLine()) != null) {

                fileStats.fileContents.add(line);

                chunk.add(line);

                if (chunk.size() >= chunkSize) {

                    List<String> chunkCopy = new ArrayList<>(chunk);

                    chunkFutures.add(contentThreadPool.submit(() -> {

                        analyzeTextChunk(chunkCopy, fileStats);

                        return null;

                    }));

                    chunk.clear();

                }

            }

            if (!chunk.isEmpty()) {

                chunkFutures.add(contentThreadPool.submit(() -> {

                    analyzeTextChunk(chunk, fileStats);

                    return null;

                }));

            }

        }

        for (Future<Void> future : chunkFutures) {

            future.get();

        }

        return fileStats;

    }

    private static void analyzeTextChunk(List<String> lines, TextStats stats) {

        Pattern pattern = Pattern.compile("[\\p{L}\\p{N}']+");

        for (String line : lines) {

            stats.lineCount++;

            stats.charCount += line.length();

            Matcher matcher = pattern.matcher(line.toLowerCase());

            Map<String, Integer> localWordCounts = new HashMap<>();

            int localWordCount = 0;

            while (matcher.find()) {

                String word = matcher.group();

                localWordCount++;

                localWordCounts.merge(word, 1, Integer::sum);

            }

            stats.wordCount += localWordCount;

            mergeWordCounts(stats.wordCounts, localWordCounts);

        }

    }

    private static void mergeWordCounts(ConcurrentHashMap<String, Integer> global, Map<String, Integer> local) {

        local.forEach((word, count) ->

                global.compute(word, (k, v) -> (v == null) ? count : v + count)

        );

    }

    private static void displayStats(TextStats stats, String label) {

        System.out.println("\n=== " + label + " ===");

        System.out.printf("Line Count: %d\n", stats.lineCount);

        System.out.printf("Character Count: %d\n", stats.charCount);

        System.out.printf("Word Count: %d\n", stats.wordCount);

        System.out.printf("Unique Word Count: %d\n", stats.wordCounts.size());

        System.out.println("\nTop 5 Most Frequent Words:");

        stats.wordCounts.entrySet().stream()

                .sorted(Map.Entry.<String, Integer>comparingByValue(Comparator.reverseOrder()))

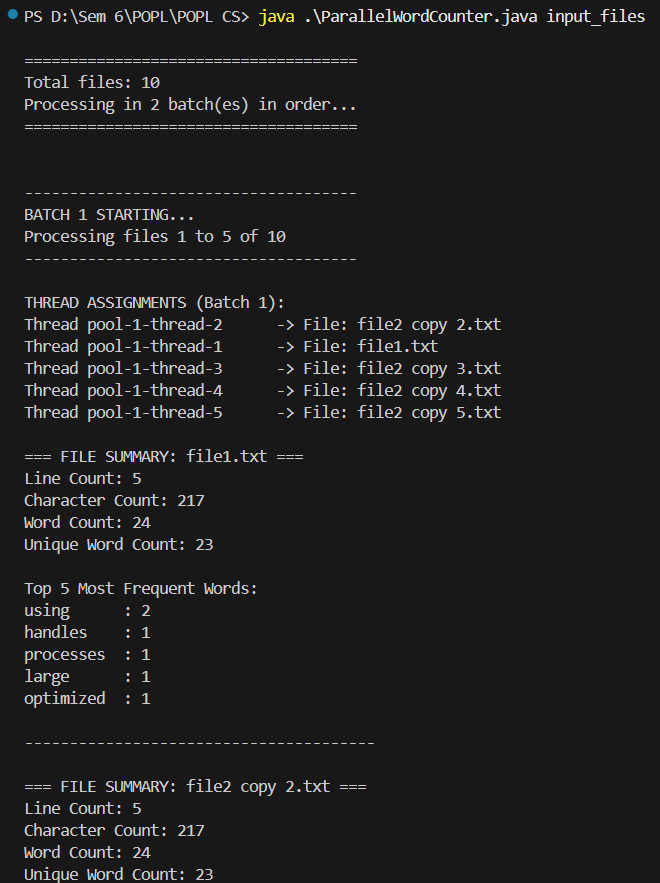
                .limit(5)

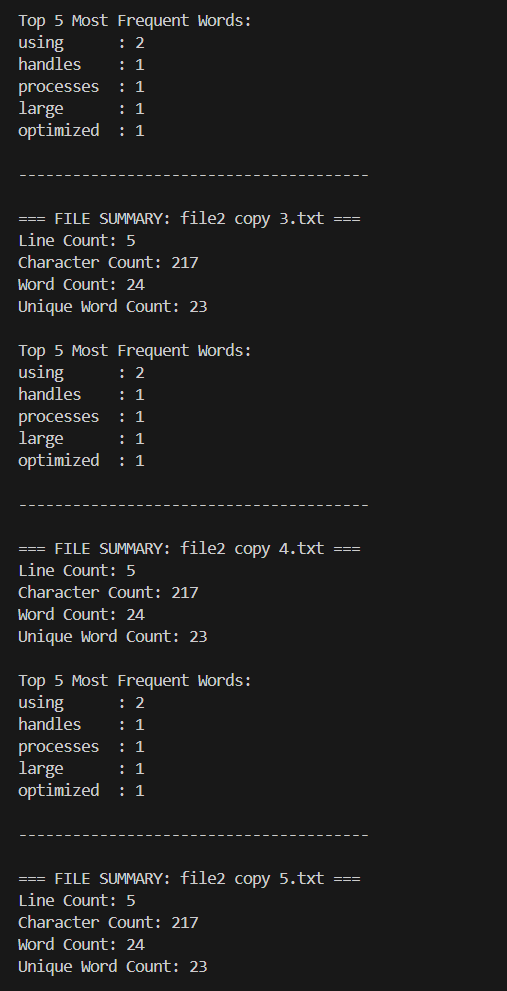
                .forEach(entry -> System.out.printf("%-10s : %d\n", entry.getKey(), entry.getValue()));

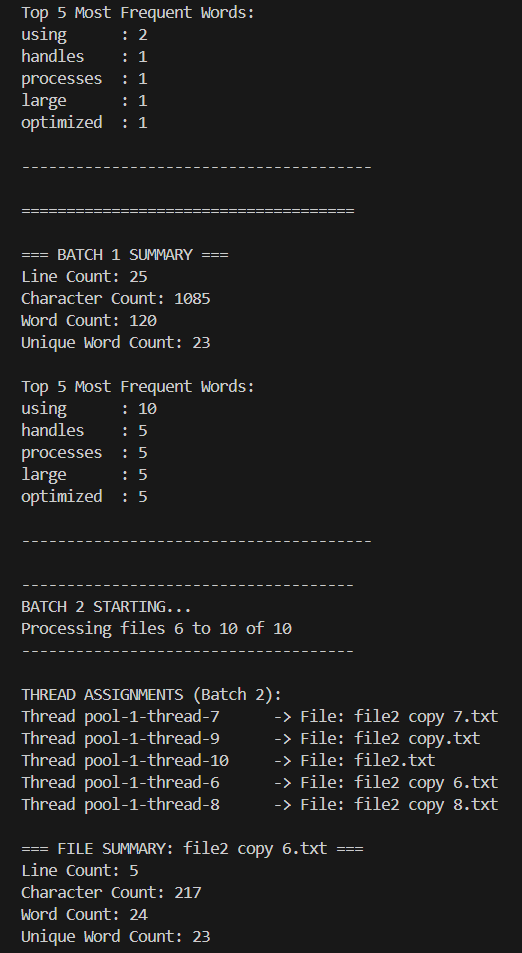
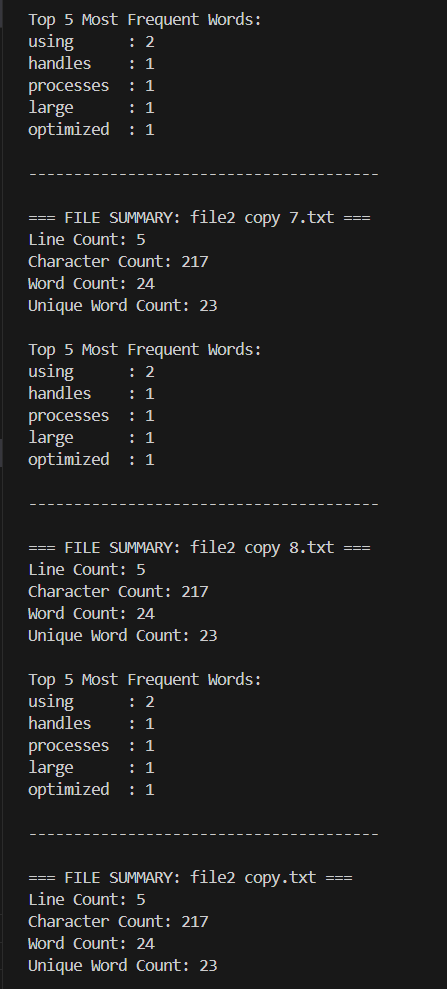
        System.out.println("\n---------------------------------------");

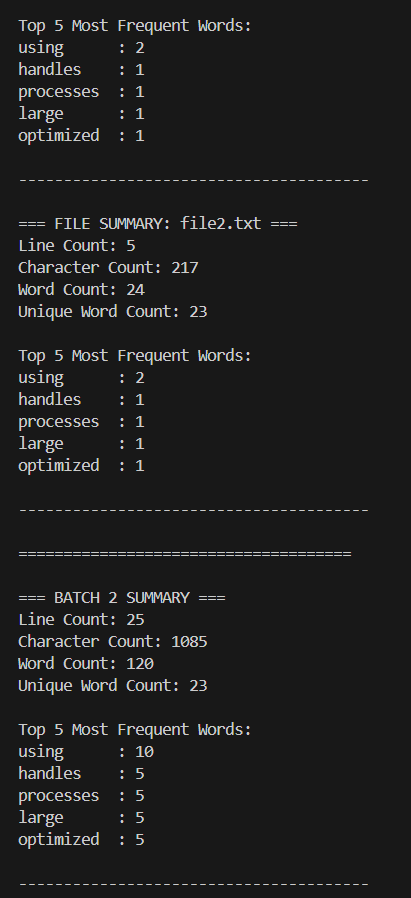
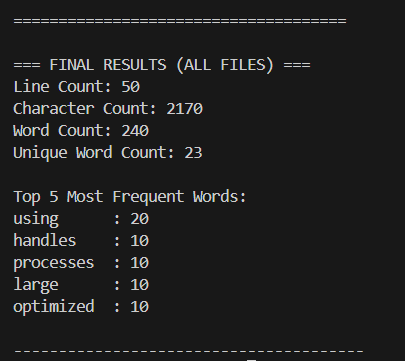
    }

}

**OUTPUT:**



** **

** **

**6. Challenges and Limitations**

While the implementation offers an efficient solution for parallel file processing, several challenges and limitations remain:

**6.1 Memory Management Challenges**

1. **File Content Storage**
   * The current implementation stores the entire content of processed files in memory
   * This approach could lead to memory exhaustion when processing large files
   * Potential solution: Implement streaming processing without storage
2. **Unbounded Collections**
   * Word frequency maps can grow indefinitely
   * Very large files with diverse vocabulary could consume substantial memory
   * Potential solution: Implement frequency thresholds or pruning strategies

**6.2 Scalability Limitations**

1. **Fixed Thread Pool Size**
   * Thread pools are sized based on available processors
   * May not be optimal for I/O-bound vs. CPU-bound operations
   * Potential solution: Implement dynamic sizing or separate pools for different operation types
2. **Fixed Chunk Size**
   * The chunk size is fixed at 100 lines
   * Not adaptive to file characteristics or system resources
   * Potential solution: Implement dynamic chunking based on file size or line length

**6.3 Error Handling Deficiencies**

1. **Limited Error Recovery**
   * Exceptions during file processing can terminate the entire batch
   * No mechanism to skip problematic files and continue processing
   * Potential solution: Implement robust error handling with partial results preservation
2. **Lack of Logging**
   * Minimal diagnostic information for troubleshooting
   * Console output is the only form of reporting
   * Potential solution: Implement comprehensive logging framework

**6.4 Configuration Inflexibility**

1. **Hardcoded Parameters**
   * Batch size and chunk size are hardcoded constants
   * No runtime configuration options
   * Potential solution: Implement command-line options or configuration files
2. **One-Size-Fits-All Approach**
   * Same processing strategy for all file types
   * No optimization for specific content patterns
   * Potential solution: Implement content-aware processing strategies

**7. Performance Considerations**

The performance of the parallel processing solution depends on several factors:

**7.1 CPU Utilization**

1. **Processor Adaptation**
   * Automatically adjusts to available CPU cores
   * Maximizes parallel processing capabilities
2. **Multi-Level Parallelism**
   * Two-tier approach (file and chunk level)
   * Ensures CPU resources are utilized efficiently

**7.2 I/O Bottlenecks**

1. **Disk I/O Limitations**
   * Multiple threads reading simultaneously can saturate disk I/O
   * Performance may be limited by storage subsystem characteristics
2. **Buffered Reading**
   * Uses BufferedReader for efficient line reading
   * Minimizes system calls for improved I/O performance

**7.3 Synchronization Overhead**

1. **Contention Points**
   * Updates to shared statistics could create contention
   * Local-to-global aggregation helps mitigate this issue
2. **Wait Operations**
   * Waiting for Future completions introduces synchronization points
   * Could limit theoretical maximum parallelism

**8. Conclusion**

The ParallelWordCounter implementation demonstrates an effective approach to concurrent text file processing, leveraging Java's thread management capabilities to achieve high throughput and efficient resource utilization. The two-tier parallelism model (files and chunks) provides a balanced approach to maximizing CPU utilization while maintaining manageable complexity.

Key strengths of the implementation include:

1. **Efficient Resource Utilization**
   * Adapts to available processors
   * Maximizes parallel processing capabilities
2. **Structured Data Management**
   * Well-organized statistics collection and aggregation
   * Thread-safe operations for consistent results
3. **Scalable Processing Approach**
   * Batch processing for manageability
   * Hierarchical result aggregation

Despite some limitations in memory management, error handling, and configuration flexibility, the solution provides a solid foundation for building sophisticated text processing applications. With the suggested enhancements, the system could evolve into an even more robust and versatile tool for large-scale text analysis.

This case study illustrates how parallel processing techniques can significantly improve performance for I/O and CPU-intensive operations, offering valuable insights for designing efficient data processing systems.