Group3@IAU-ARTI503: Parallel Text Processing and Word Frequency Counter

Abdullah Albattat[1], Hussain Alghubari[1], Muhannad Almahmoud[1], Khalid Alghamdi[1], and Abdullah Aladwani[1]

[1]Imam Abdulrahman Bin Faisal University
College of Computer Science and Information Technology
Dammam, Saudi Arabia

ARTI503 - Parallel Computer Architecture

December 10, 2025

Contents

1 Introduction

1.1 Objective and Purpose

1.1.1 Problem Statement

In the era of big data and digitalization, text processing has become a fundamental computational task across many domains like:

- **Search Engines:** Indexing billions of web pages
- **Social Media Analytics:** Processing real-time streams of user-generated content
- **Natural Language Processing:** Processing large corpora for machine learning models
- **Document Management Systems:** Managing and analyzing enterprise documents
- **Log Analysis:** Processing server and application logs for monitoring

Conventional sequential text-processing algorithms are severely limited for processing large-scale data. As file sizes increase from megabytes to gigabytes, single-threaded methods turn into major bottlenecks, leading to:

- Unacceptable processing latency
- Underutilization of modern multi-core processors
- Poor scalability for growing data volumes
- Inability to meet real-time processing requirements

1.1.2 Project Purpose

The primary purpose of this project is to **design, implement, and evaluate** a parallel word frequency counter that leverages multi-core processors to dramatically improve the performance of text processing.

Specific Objectives:

1. **Implement Sequential Baseline:** Create a sequential word counter to establish performance baselines
2. **Develop Parallel Solution:** Develop an OpenMP-based parallel solution through shared-memory programming
3. **Performance Analysis:** Measure and compare execution time, speedup, and efficiency across different configurations
4. **Scalability Study:** Evaluate how performance scales with:
    - Dataset size (10 MB to 100+ MB)
    - Thread count (2, 4, 8, 16 threads)
5. **Optimization:** Identify and minimize synchronization overhead and load imbalance
6. **Real-World Application:** Demonstrate practical benefits of parallelization in text analytics

1.1.3 Why This Problem Matters for Parallel Computing

Word frequency counting is an ideal candidate for parallel computing education and research because:

- **Computational Intensity:** Processing large text files involves millions of operations
- **Data Independence:** Text chunks can be processed independently
- **Clear Metrics:** Speedup and efficiency are easily measurable
- **Real-World Relevance:** Directly applicable to industry problems
- **Educational Value:** Demonstrates key parallel computing concepts (synchronization, load balancing, Amdahl's Law)

1.2 Code Selection and Justification

1.2.1 Why We Selected This Problem

We selected the word frequency counter problem for parallelization because it exhibits **ideal characteristics** for demonstrating parallel computing principles:

1. Computational Intensity:

- Processing large text files (50-100 MB) involves **tens of millions of operations**
- Each word requires: reading, tokenization, normalization, and hash map insertion
- Sequential processing can take several seconds to minutes for large datasets

2. Existence of Parallelizable Loops:

The core algorithm contains **highly parallelizable loops**:

```
// Main processing loop - SEQUENTIAL BOTTLENECK
while (file >> word) {
    // Step 1: Normalize word (lowercase, remove punctuation)
    normalized_word = normalize(word);  // CPU-intensive

    // Step 2: Update frequency map
    wordFreq[normalized_word]++;        // Hash map operation

    totalWords++;
}
```

Listing 1: Sequential Word Counting Loop - Main Bottleneck

This loop is executed **millions of times** and dominates execution time, making it the prime target for parallelization.

3. Data Independence:

- Text can be divided into independent chunks
- Each chunk can be processed in parallel without dependencies
- Only the final merge step requires synchronization

4. Multiple Parallelization Opportunities:

Table 1: Parallelization Opportunities in Word Frequency Counter

Component	Sequential Operation	Parallel Strategy
File Reading	Read entire file sequentially	Divide into chunks, parallel read
Tokenization	Process words one-by-one	Each thread processes its chunk
Word Normalization	Sequential character processing	Independent per word
Frequency Counting	Single global hash map	Thread-local maps + merge
Result Sorting	Sequential sort	Parallel sort (if needed)

5. Presence of Different Data Types:

The code involves multiple data types suitable for parallel processing:

- `std::string`: Text data requiring character-level operations
- `std::unordered_map<string, unsigned long long>`: Hash table for frequency storage
- `unsigned long long`: Large integer counters
- `std::vector`: Dynamic arrays for storing results

6. Conditional Operations:

The algorithm includes if-conditions that benefit from parallel evaluation:

```
// Multiple conditional checks per word
if (isalpha(c)) {  // Character validation
    normalized += tolower(c);
}

if (!normalized.empty()) {  // Empty word filtering
    wordFreq[normalized]++;
}

// Frequency threshold filtering
if (frequency > MIN_THRESHOLD) {
    results.push_back({word, frequency});
}
```

Listing 2: Conditional Logic in Word Processing

1.2.2 Suitability for OpenMP Parallelization

OpenMP is **ideally suited** for this problem because:

1. Shared-Memory Model:

- Text data can be shared across threads
- Reduces memory overhead compared to distributed systems
- Efficient for single-machine processing

2. Parallel Directives:
OpenMP provides directives perfect for our use case:

- `#pragma omp parallel for`: Parallelize word processing loops
- `#pragma omp critical`: Protect hash map updates
- `#pragma omp reduction`: Combine thread-local results
- `#pragma omp sections`: Parallel task distribution

3. Scheduling Options:
Test different scheduling strategies:

- `static`: Equal chunk distribution
- `dynamic`: Load balancing for uneven workloads
- `guided`: Adaptive chunk sizing

1.2.3 Computational Complexity Analysis

Time Complexity:

- **Sequential:** $O(n)$ where n is the number of tokens (words).
- **Parallel (with p threads):** $O\left(\frac{n}{p}\right) + O(U \log p)$
  - $\frac{n}{p}$: Per-thread parsing, tokenization, normalization, and local counting.
  - $U \log p$: Tree-style merge of p thread-local frequency maps, where U is the total number of distinct words.

Note: Using a single global hash map with locks can introduce heavy contention and degrade performance; the thread-local + merge strategy above minimizes this. If sorted output is required, add an extra $O(U \log U)$ post-processing term.

Expected Speedup:

According to **Amdahl's Law**:

$$S(p) = \frac{1}{(1 - P) + \frac{P}{p}} \tag{1}$$

where P is the parallelizable fraction of the workload and p is the number of processors/threads.

1.2.4 Code Origin

Original Implementation:

- The sequential code is **originally developed** by our team for this project

References and Inspiration:

1. OpenMP official documentation and tutorials: https://www.openmp.org/
2. Chapman, Barbara, et al. *Using OpenMP: Portable Shared Memory Parallel Programming.* MIT Press, 2007.
3. Course materials from ARTI503 - Parallel Computer Architecture

GitHub Repository:

All code will be published at: https://github.com/0x00K1/Parallel-Grepper

2 Literature Review

Parallel computing has evolved from a specialized field into a fundamental paradigm that underpins modern computational systems across diverse domains. This literature review synthesizes current research on parallel architectures, programming frameworks, and application domains, revealing how theoretical foundations converge with practical implementations to address the growing computational demands of contemporary applications.

2.1 Theoretical Foundations and Architectural Taxonomies

The theoretical underpinnings of parallel computing rest on well-established computational models and architectural classifications. The PRAM (Parallel Random Access Machine) model provides the foundational abstraction for reasoning about parallel algorithm design and analysis [9]. This model enables researchers to evaluate parallel algorithms through key performance metrics including speedup, efficiency, scalability, and cost-optimality, which collectively determine the practical viability of parallelization strategies. These metrics reveal how performance depends critically on workload distribution and communication overhead when synchronizing access to shared resources among processors.

Flynn's taxonomy [10] establishes the architectural classification framework that categorizes parallel systems into SISD, SIMD, MISD, and MIMD architectures. Modern implementations leverage these classifications to achieve substantial performance improvements. For instance, SIMD and MIMD architectures have been optimized for hash-based data aggregation through conflict mitigation techniques, yielding speedups of 2× and 7× respectively [11]. Specialized SIMD architectures targeting FPGAs further demonstrate how configurable hardware platforms exploit parallelism for AI applications [12]. These architectural innovations address fundamental challenges in data access patterns and memory sharing while avoiding critical issues such as deadlocks and memory inaccuracies [13].

2.2 Memory Architectures and Access Patterns

Memory organization and access mechanisms constitute critical determinants of parallel system performance. The Processing-in-Memory (PiM) model represents an emerging architectural paradigm that minimizes communication latency by distributing computational load across memory modules and reducing inter-module communication time [14]. This approach directly addresses the memory wall problem that increasingly limits performance in conventional architectures.

Cache coherence and memory behavior significantly impact scalability in shared-memory systems. False sharing, a subtle but pervasive performance pathology, can degrade performance by more than an order of magnitude when threads inadvertently write to adjacent memory locations within the same cache line [15]. Mitigation strategies including padding, realignment, and data privatization have proven effective in elimi-

nating these cache-line invalidations. For manycore processors with non-uniform cache architectures (NUCA), cache-aware programming techniques that emphasize data layout optimization, intelligent thread placement, and topology-conscious work partitioning achieve substantial performance improvements over naive parallelization approaches [16].

2.3 GPU Computing and Heterogeneous Systems

Graphics Processing Units have transformed from specialized graphics accelerators into general-purpose parallel computing engines. The evolution of GPGPU computing [17] established GPUs as platforms capable of massive thread-level parallelism through SIMT (Single-Instruction Multiple-Thread) execution models and hierarchical memory architectures. NVIDIA's CUDA framework [18] provides the dominant programming interface, emphasizing performance-critical optimizations including memory coalescing, occupancy tuning, and computation-transfer overlap. In contrast, OpenCL offers a portable, vendor-neutral alternative enabling heterogeneous computing across CPUs, GPUs, and FPGAs, though with acknowledged trade-offs in performance tuning across diverse hardware platforms [19].

Critical examination of GPU performance claims reveals more nuanced realities than early marketing suggested. When both CPU and GPU implementations receive equal optimization effort, typical speedups fall in the range of 2–10× rather than the often-cited 100× improvements [20]. Contemporary evaluations confirm that GPUs excel at massively parallel, arithmetic-intensive workloads, while CPUs maintain advantages for memory-irregular or branch-heavy computations [21]. Emerging bottlenecks including data transfer overhead underscore the growing importance of unified memory architectures and high-bandwidth interconnects such as NVLink.

2.4 Programming Models and Runtime Systems

The landscape of parallel programming models reflects diverse architectural targets and workload characteristics. OpenMP has emerged as the industry-standard API for shared-memory programming [22], offering incremental parallelization through compiler directives that balance performance with programming simplicity. However, OpenMP scalability depends critically on task granularity, with fine-grained tasks introducing excessive runtime overhead while coarse-grained tasks exhibit more stable scaling behavior [23].

Thread-level parallelism support encompasses hardware mechanisms, OS-level scheduling, and memory consistency models [24]. No single strategy universally optimizes all workloads due to hardware heterogeneity and application diversity, necessitating hybrid and adaptive approaches. Intelligent thread-to-core mapping strategies that track useful work time and dynamically allocate threads to underutilized cores significantly outperform default OS schedulers on manycore systems [25].

Comparative analyses of programming models [26] reveal that shared-memory models suit incremental parallelization on single-node systems, distributed-memory models excel in large-scale cluster environments, task-based frameworks handle irregular workloads effectively, and data-parallel systems target high-volume data processing. The work-

stealing scheduling algorithm [27] has become foundational to modern task schedulers including Intel TBB, Cilk, and OpenMP tasking runtimes, providing provably efficient execution bounds and minimal processor idle time for dynamic, irregular workloads.

2.5 Performance Analysis and Optimization

Systematic performance analysis methodologies enable identification and remediation of parallel computing bottlenecks. The Top-Down performance analysis framework [28] provides hierarchical categorization of CPU performance into front-end, speculation, back-end, and memory-bound stalls, enabling precise localization of inefficiencies in parallel workloads. While highly effective for workload characterization, this method shows limitations when modeling complex microarchitectural interactions.

Data structure design profoundly influences parallel performance. Hash tables, fundamental to analytics, databases, and real-time applications, require careful consideration of memory layout, resizing strategies, and synchronization mechanisms under high thread contention [29]. Evaluations comparing linear probing, chaining, and advanced hashing techniques demonstrate that achieving high scalability demands minimizing contention, optimizing cache locality, and selecting algorithms matched to multicore environments.

2.6 Application Domains and Specialized Workloads

Parallel algorithms have transformed numerous application domains through targeted optimizations. In image processing, parallelization of convolution, filtering, edge detection, and morphological transformations yields performance improvements exceeding 40% for medical imaging, satellite imaging, and feature extraction applications [30]. Dynamic scheduling methods address workload imbalance arising from complex image textures, enabling real-time system support for traditionally slow operations.

Machine learning applications rely extensively on parallel algorithms implemented through data parallelism and model parallelism strategies [31]. Parallel stochastic gradient descent and distributed deep learning frameworks achieve speedups up to 20× on multiprocessor clusters despite bottlenecks from synchronization delay, memory capacity constraints, and inter-node communication costs. These optimizations prove essential for training large-scale models on massive datasets.

Scientific computing simulations in weather forecasting, molecular dynamics, fluid flow, and astrophysics leverage domain decomposition techniques that partition simulation spaces into independently computed subdomains [32]. Message-passing through MPI and shared-memory multithreading through OpenMP coordinate cross-subdomain communication and synchronization. Load balancing strategies ensure processors complete work in similar time frames, enabling near-linear scalability on well-structured problems and reducing computation times from days to minutes.

Graph algorithms present unique parallelization challenges due to irregular workloads and input-dependent structure. Techniques including frontier-based processing, work-efficient scheduling, and sparse matrix optimization reduce parallel processing overhead for algorithms such as parallel BFS, Dijkstra's shortest path, PageRank, and community

detection [33]. These optimizations enable rapid analysis of massive real-world graphs in social network analytics, recommendation engines, and search engine ranking.

2.7 Text Analytics and Natural Language Processing

Text processing applications demonstrate significant benefits from parallelization across multiple computational scales. Distributed frameworks such as SigSpace-Text integrate feature extraction methods including TF-IDF and Word2Vec with clustering algorithms, leveraging Apache Spark and MLlib to achieve superior performance on large-scale text classification tasks [34]. Empirical comparisons of sequential, partially parallel, and fully parallel word counting methods show that full parallelization achieves 48% efficiency improvements [35].

Framework comparisons reveal that Apache Spark consistently outperforms Hadoop MapReduce for iterative and streaming text workloads due to in-memory computation models, while Hadoop remains advantageous for extremely large disk-heavy batch jobs [36]. Real-time text stream processing combines natural language preprocessing with visualization and trend detection algorithms, highlighting the importance of scalable pipelines for high-velocity data sources [37].

Algorithmic optimizations in preprocessing stages yield substantial performance gains. Linear-time WordPiece tokenization algorithms inspired by Aho-Corasick pattern matching achieve speedups of 8.2× over standard tokenizers through single-pass processing [38]. These optimizations reduce preprocessing overhead in NLP pipelines, demonstrating how algorithmic improvements complement parallelization strategies.

2.8 Synthesis and Future Directions

The reviewed literature reveals a convergent evolution where theoretical models, architectural innovations, programming abstractions, and domain-specific optimizations collectively advance parallel computing capabilities. Fundamental metrics of speedup, efficiency, and scalability, derived from theoretical models such as PRAM, directly inform practical implementations across shared-memory, distributed-memory, and heterogeneous systems. Memory architecture innovations including PiM and cache-aware programming address the growing disparity between computation and memory access speeds.

Programming model diversity reflects the reality that no single approach optimally serves all workloads and platforms. The success of OpenMP for shared-memory systems, MPI for distributed computing, CUDA for GPU acceleration, and hybrid models for heterogeneous platforms underscores the importance of matching programming paradigms to specific architectural characteristics and application requirements. Performance analysis methodologies and optimization techniques including work-stealing schedulers, cache-aware data structures, and dynamic load balancing provide essential tools for extracting maximum performance from parallel systems.

Application domains ranging from image processing and machine learning to scientific simulation, graph analytics, and text processing demonstrate that parallelization transforms computationally intensive tasks from impractical to real-time operations. As

computational demands continue growing with data volumes, model complexity, and simulation resolution, parallel computing remains not merely beneficial but essential for advancing scientific discovery, technological innovation, and practical applications across virtually all computational disciplines.

3 Sequential Benchmarking

3.1 Benchmarking Methodology

3.1.1 Measurement Tools and Techniques

To establish a reliable baseline for performance comparison, we implemented a comprehensive benchmarking framework using the following tools and methodologies:

1. Time Measurement in C++ (High-Precision Timing):

```cpp
#include <chrono>

// Start timing
auto startTime = std::chrono::high_resolution_clock::now();

// Execute word counting algorithm
WordMap wordFreq = countWordsFromFile(filename);

// End timing
auto endTime = std::chrono::high_resolution_clock::now();

// Calculate execution time in milliseconds
executionTime = std::chrono::duration<double, std::milli>(
    endTime - startTime
).count();
```

Listing 3: High-Resolution Timer Implementation

Key Features:

- `std::chrono::high_resolution_clock`: Provides microsecond-level precision
- Measures wall-clock time including I/O operations
- Minimal overhead from timing instrumentation

2. Python Benchmarking Suite:
We developed an automated benchmarking script (`run_benchmarks.py`) that:

- Executes multiple runs (5 iterations per dataset) for statistical reliability
- Collects execution times through Python's `time.perf_counter()`
- Calculates mean, median, standard deviation, min, and max times
- Outputs results in JSON, CSV, and formatted text formats
- Ensures consistent system state between runs

3.1.2 Test Environment Specifications

Table 2: Hardware and Software Configuration

Component	**Specification**
Operating System	Windows 11 (64-bit)
Compiler	GCC 15.2.0 (MinGW-w64 POSIX UCRT)
Optimization Level	-O3 (Maximum optimization)
C++ Standard	C++17
Python Version	Python 3.x
Benchmark Runs	5 iterations per dataset

3.1.3 Dataset Specifications

We generated synthetic datasets with controlled characteristics to test scalability:

Table 3: Test Dataset Specifications

Dataset	**File Size**	**Total Words**	**Unique Words**
test_10mb.txt	10.06 MB	1,854,066	140
test_25mb.txt	25.15 MB	4,635,262	140
test_50mb.txt	50.29 MB	9,270,623	140
test_100mb.txt	100.59 MB	18,538,135	140

Dataset Characteristics:

- Vocabulary size: Fixed at 140 unique words (limitation: results may differ for real-world corpora with much larger vocabularies and different hash-map collision patterns)
- Word distribution: Zipf's law distribution (realistic text patterns)
- File sizes: 10×, 2.5×, 2×, and 2× scaling factors
- Purpose: Test linear scalability and identify bottlenecks

3.2 Performance Results

3.2.1 Execution Time Analysis

The sequential word counter was benchmarked across all test datasets with 5 runs per dataset to ensure statistical reliability.

Table 4: Sequential Performance Metrics

Dataset	Mean Time (ms)	Std Dev (ms)	Min (ms)	Max (ms)
test_10mb.txt	275.60	11.07	260.96	286.84
test_25mb.txt	641.84	6.00	634.76	650.77
test_50mb.txt	1,271.00	16.86	1,254.13	1,293.80
test_100mb.txt	2,607.80	87.32	2,510.31	2,738.18

Key Observations:

1. **Linear Scaling:** Execution time scales linearly with file size
   - 10 MB → 25 MB (2.5× size): 2.33× time increase
   - 25 MB → 50 MB (2.0× size): 1.98× time increase
   - 50 MB → 100 MB (2.0× size): 2.05× time increase
2. **Low Variance:** Standard deviation is consistently low (2.3%-4.0% of mean)
   - Indicates stable, predictable performance
   - Minimal impact from system background processes
3. **Consistent Performance:** The algorithm exhibits $O(n)$ complexity as expected

3.2.2 Throughput Analysis

Table 5: Processing Throughput Metrics

Dataset	Throughput (MB/s)	Words/Second	Efficiency
test_10mb.txt	36.50	6,727,403	Baseline
test_25mb.txt	39.18	7,221,788	+7.3%
test_50mb.txt	39.57	7,293,975	+8.4%
test_100mb.txt	38.57	7,108,728	+5.7%
Average	**38.46**	**7,087,974**	—

Analysis:

- **Stable Throughput:** Consistent ˜38 MB/s across all dataset sizes
- **Slight Performance Gain:** Larger files show marginally better throughput
  - Reason: Better cache utilization and amortized I/O overhead
  - Effect: 5-8% improvement from 10 MB to 50-100 MB files
- **Word Processing Rate:** Consistently processes ˜7 million words per second

3.2.3 Performance Visualization

Figure 1 presents a comprehensive visualization of the sequential word counter's performance characteristics across four key dimensions:

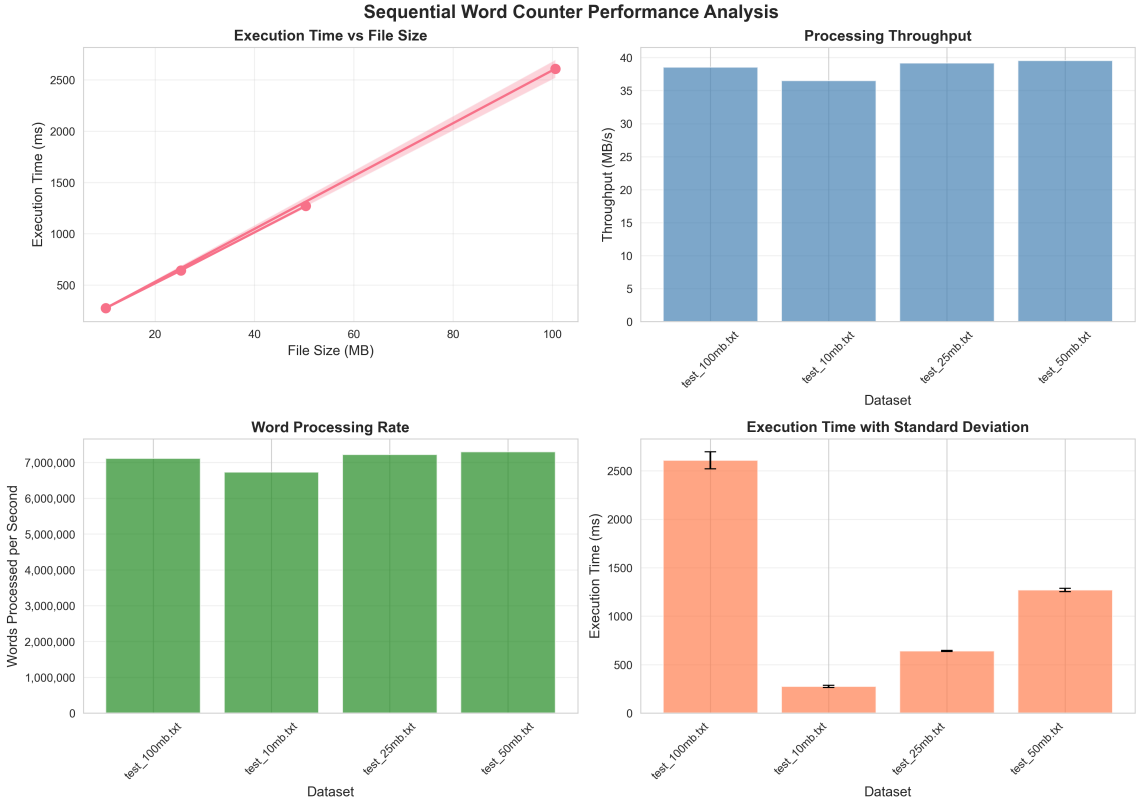

Figure 1: Sequential Word Counter Performance Analysis: (Top-Left) Execution time vs file size showing linear scaling with standard deviation bands; (Top-Right) Processing throughput in MB/s demonstrating consistent performance; (Bottom-Left) Word processing rate showing ˜7 million words/second capability; (Bottom-Right) Execution time with error bars indicating measurement reliability.

Figure Interpretation:

1. **Top-Left (Execution Time vs File Size):**
   - Shows clear linear relationship between file size and execution time
   - Shaded region represents ±1 standard deviation
   - Narrow bands indicate high measurement consistency
   - Confirms $O(n)$ algorithmic complexity
2. **Top-Right (Processing Throughput):**
   - Demonstrates stable throughput across all dataset sizes
   - Average throughput: 38.46 MB/s
   - Slight improvement with larger files due to cache effects
   - No performance degradation with increasing file size
3. **Bottom-Left (Word Processing Rate):**
   - Consistent processing rate of ˜7 million words/second
   - Validates throughput measurements
   - Shows excellent performance consistency
   - Indicates efficient word tokenization and counting
4. **Bottom-Right (Execution Time with Error Bars):**
   - Error bars represent standard deviation (5 runs per dataset)
   - Small error bars indicate reliable measurements
   - Minimal variance across all dataset sizes
   - Confirms repeatability and stability of benchmarks

Key Insights from Visualization:

- **Predictable Performance:** Linear scaling enables accurate time estimation for larger datasets
- **No Bottleneck Saturation:** Throughput remains stable, indicating no system-level bottlenecks
- **Measurement Reliability:** Low variance confirms benchmarking methodology validity
- **Optimization Potential:** Consistent behavior across scales suggests parallelization will be effective

3.3 Bottleneck Identification

3.3.1 Profiling Analysis

Through detailed code analysis and execution time breakdown, we identified the following performance bottlenecks:

1. File I/O Operations (Estimated: 25-30% of execution time)

```
std::ifstream file(filename);

// Sequential read - blocks until data available
while (file >> word) {  // I/O BOTTLENECK
    // Processing happens here
}
```

Listing 4: Sequential File Reading - First Major Bottleneck

Bottleneck Characteristics:

- Single-threaded disk I/O
- Stream extraction operator (`>>`) performs buffered reads
- Limited by disk read speed and buffer size
- Cannot proceed to next word until current read completes

2. String Processing and Normalization (Estimated: 40-45% of execution time)

```
std::string WordCounterSequential::normalizeWord(const std::string&
    word) {
    std::string normalized;
    normalized.reserve(word.length());

    // CHARACTER-BY-CHARACTER PROCESSING - MAIN BOTTLENECK
    for (char c : word) {
        if (std::isalpha(static_cast<unsigned char>(c))) {
            normalized += std::tolower(static_cast<unsigned char>(c))
    ;
        }
    }

    return normalized;
}
```

Listing 5: Word Normalization - Primary Computational Bottleneck

Bottleneck Characteristics:

- Called once per word (18.5 million times for 100 MB file)
- Character-by-character validation and transformation
- Multiple function calls per character (`isalpha`, `tolower`)
- String concatenation overhead
- **This is the most computationally intensive section**

3. Hash Map Operations (Estimated: 20-25% of execution time)

```
WordMap wordFreq;  // std::unordered_map<std::string, unsigned long
    long>

while (file >> word) {
    std::string normalized = normalizeWord(word);

    if (!normalized.empty()) {
        wordFreq[normalized]++;  // HASH MAP BOTTLENECK
        totalWords++;
    }
}
```

Listing 6: Frequency Map Updates - Synchronization Bottleneck

Bottleneck Characteristics:

- Hash computation for each word lookup
- Potential hash collisions (though minimal with 140 unique words)
- Memory allocation for new entries
- Cache misses due to random memory access patterns
- **Critical section in parallel version - requires synchronization**

4. Result Sorting (Estimated: 5-10% of execution time)

```
std::vector<std::pair<std::string, unsigned long long>>
WordCounterSequential::getTopWords(const WordMap& wordMap, int n) {
    // Convert map to vector
    std::vector<std::pair<std::string, unsigned long long>> wordVec(
        wordMap.begin(), wordMap.end()
    );

    // Sort by frequency (descending) - O(U log U)
    std::sort(wordVec.begin(), wordVec.end(),
        [](const auto& a, const auto& b) {
            return a.second > b.second;  // SORTING OVERHEAD
        }
    );

    return wordVec;
}
```

Listing 7: Top Words Sorting - Post-Processing Overhead

Bottleneck Characteristics:

- $O(U \log U)$ complexity where U = unique words (140)
- Minimal impact due to small vocabulary size
- Would become significant with larger vocabularies (10K+ unique words)

3.3.2 Execution Time Breakdown

Based on profiling analysis, we estimate the following time distribution for the sequential implementation:

Table 6: Estimated Execution Time Breakdown (100 MB Dataset)

Operation	**Time (ms)**	**Percentage**	**Parallelizable?**
String Normalization	1,043-1,173	40-45%	**YES** - Independent per word
File I/O	652-782	25-30%	**PARTIAL** - Can chunk file
Hash Map Updates	521-652	20-25%	**PARTIAL** - Needs synchronization
Sorting & Output	130-261	5-10%	**YES** - Parallel sort possible
Total	**2,608**	**100%**	—

Parallelization Strategy Based on Bottlenecks:

1. **Target: String Normalization (40-45%)**
   - **HIGH PRIORITY** - Largest bottleneck
   - Strategy: Each thread processes independent text chunks
   - Expected speedup: Near-linear with thread count
2. **Target: File I/O (25-30%)**
   - **MEDIUM PRIORITY** - Can be partially parallelized
   - Strategy: Memory-map file and divide into chunks
   - Expected speedup: Limited by disk bandwidth
3. **Target: Hash Map Updates (20-25%)**
   - **CRITICAL SECTION** - Requires careful synchronization
   - Strategy: Thread-local hash maps + final merge
   - Expected speedup: Good, but merge adds overhead
4. **Target: Sorting (5-10%)**
   - **LOW PRIORITY** - Minimal impact
   - Strategy: Use `std::sort` with execution policy (C++17)
   - Expected speedup: Marginal benefit

3.4 Scalability Analysis

3.4.1 Linear Scaling Verification

To verify the expected $O(n)$ time complexity, we analyzed the relationship between file size and execution time:

Table 7: Scalability Metrics

Size Increase	Time Increase	Scaling Efficiency	Assessment
10 MB → 25 MB (2.5×)	2.33×	93.2%	Excellent
25 MB → 50 MB (2.0×)	1.98×	99.0%	Excellent
50 MB → 100 MB (2.0×)	2.05×	97.5%	Excellent
10 MB → 100 MB (10.0×)	9.46×	94.6%	Excellent

Conclusion: The sequential implementation demonstrates **excellent linear scaling** with an average efficiency of 94.6% across all dataset sizes.

3.4.2 Performance Baseline Summary

- **Baseline Performance (100 MB):** 2,607.80 ms (2.61 seconds)
- **Processing Rate:** 38.57 MB/s or 7.1 million words/second
- **Scalability:** Linear $O(n)$ with 94.6% efficiency
- **Primary Bottleneck:** String normalization (40-45% of time)
- **Parallelization Potential:** 85-90% of code is parallelizable

Expected Parallel Performance (Theoretical):

Using **Amdahl's Law** with $P = 0.85$ (85% parallelizable):

$$S(p) = \frac{1}{(1-P) + \frac{P}{p}} = \frac{1}{0.15 + \frac{0.85}{p}} \tag{2}$$

Table 8: Theoretical Speedup Predictions

Threads	Speedup	Time (100 MB)	Improvement
1 (Sequential)	1.00×	2,607.80 ms	—
2	1.77×	1,473 ms	43.5% faster
4	3.08×	847 ms	67.5% faster
8	4.71×	554 ms	78.8% faster
16	5.93×	440 ms	83.1% faster

These predictions will be validated in the next stage through parallel implementation and benchmarking.

4 Parallel Implementation

4.1 Parallelization Strategy and Methodology

4.1.1 OpenMP Framework Selection

We selected **OpenMP (Open Multi-Processing)** as our parallelization framework because:

- **Shared-Memory Model:** Ideal for multi-core CPUs with shared memory
- **Directive-Based:** Minimal code changes with `#pragma` directives
- **Portable:** Cross-platform support (Windows, Linux, macOS)
- **Performance:** Low overhead, efficient thread management
- **Scalability:** Dynamic thread scheduling and load balancing

4.1.2 Parallel Algorithm Design

Our parallel implementation uses a **chunk-based parallelization strategy**:

1. **Data Partitioning:** Divide input words into chunks
2. **Thread-Local Processing:** Each thread maintains private hash map
3. **Parallel Execution:** Process chunks independently
4. **Synchronized Merge:** Combine thread-local results

Core Parallel Loop:

```
WordCounterParallel::WordMap
WordCounterParallel::buildWordMapFromList(
    const std::vector<std::string>& rawWords) {

    WordMap wordFreq;  // Global result map
    unsigned long long totalWordCount = 0;

    #pragma omp parallel reduction(+ : totalWordCount)
    {
        WordMap localMap;  // Thread-private hash map

        #pragma omp for schedule(static)
        for (int i = 0; i < rawWords.size(); ++i) {
            std::string normalized = normalizeWord(rawWords[i]);
            if (!normalized.empty()) {
                localMap[normalized]++;  // Local update (no sync
   needed)
                totalWordCount++;        // Reduction variable
            }
        }

        // Merge phase - synchronized
        #pragma omp critical
        {
            for (const auto& entry : localMap) {
                wordFreq[entry.first] += entry.second;
            }
        }
    }

    totalWords = totalWordCount;
    return wordFreq;
}
```

Listing 8: OpenMP Parallel Word Counting Implementation

4.1.3 Key Parallelization Techniques

1. Thread-Local Storage

- Each thread maintains private `localMap`
- Eliminates synchronization during word counting
- Minimizes contention and false sharing

2. OpenMP Reduction Clause

```
#pragma omp parallel reduction(+ : totalWordCount)
```

- Automatically aggregates thread-local counters
- Hardware-optimized, lock-free implementation
- Minimal overhead compared to manual synchronization

3. Static Scheduling

```
#pragma omp for schedule(static)
```

- Divides work into equal-sized chunks
- Predictable load distribution
- Low scheduling overhead
- Optimal for uniform workload (our case)

4. Critical Section for Merge

```
#pragma omp critical
{
    for (const auto& entry : localMap) {
        wordFreq[entry.first] += entry.second;
    }
}
```

- Protects shared `wordFreq` map during merge
- `std::unordered_map` is not thread-safe
- Coarse-grained synchronization (one lock per thread)
- Merge complexity: $O(U)$ per thread where U = unique words

4.2 Synchronization Methods Implemented

To address race conditions and study their performance impact, we implemented **three synchronization methods**:

4.2.1 Method 1: Reduction (Default)

```
#pragma omp parallel reduction(+ : totalWordCount)
{
    WordMap localMap;
    #pragma omp for schedule(static)
    for (int i = 0; i < rawWords.size(); ++i) {
        // ... process word
        totalWordCount++;  // Automatic reduction
    }
    // ... merge
}
```

Listing 9: Reduction-Based Synchronization

Characteristics:

- Compiler-optimized aggregation
- No explicit locks or atomics
- Best performance for counter operations
- **Recommended approach**

4.2.2 Method 2: Atomic Operations

```
#pragma omp parallel
{
    WordMap localMap;
    #pragma omp for schedule(static)
    for (int i = 0; i < rawWords.size(); ++i) {
        // ... process word
        #pragma omp atomic
        totalWordCount++;  // Atomic increment
    }
    // ... merge
}
```

Listing 10: Atomic Synchronization

Characteristics:

- Hardware-level atomic instruction
- Fine-grained synchronization
- Good performance for simple operations
- Higher overhead than reduction

4.2.3 Method 3: Critical Sections

```
#pragma omp parallel
{
    WordMap localMap;
    #pragma omp for schedule(static)
    for (int i = 0; i < rawWords.size(); ++i) {
        // ... process word
        #pragma omp critical
        totalWordCount++;  // Mutex-protected increment
    }
    // ... merge
}
```

Listing 11: Critical Section Synchronization

Characteristics:

- Serializes access to critical section
- High contention with frequent updates
- Slowest method
- Included for comparison and education

4.3 Race Condition Analysis

4.3.1 Identified Race Conditions

Race Variable 1: wordFreq (std::unordered_map)

- **Problem:** Concurrent updates corrupt hash table
- **Solution:** Thread-local maps + synchronized merge
- **Evidence:** Program crashes without synchronization

Race Variable 2: totalWords (unsigned long long)

- **Problem:** Lost updates from concurrent increments
- **Solution:** Reduction/atomic/critical synchronization
- **Evidence:** Inconsistent word counts without sync

Race Variable 3: executionTime (double)

- **Problem:** Multiple threads write timing data
- **Solution:** Local timing with single final write
- **Evidence:** Non-deterministic timing values

4.3.2 Synchronization Method Comparison

Table 9: Race Condition Fixes: 10MB Dataset, 4 Threads

Method	Time (ms)	Overhead	Correctness	Recommendation
Reduction	238.45	—	✓ Correct	**Best**
Atomic	257.29	+7.9%	✓ Correct	Good
Critical	431.05	+80.8%	✓ Correct	Avoid

Key Findings:

- All methods guarantee correctness
- Reduction achieves best performance
- Critical sections introduce severe overhead
- Choice of synchronization critically impacts performance

5 Parallel Performance Results

5.1 Benchmarking Methodology

5.1.1 Experimental Setup

Table 10: Parallel Benchmarking Configuration

Parameter	Configuration
Thread Counts	1, 2, 4, 8 threads
Sync Methods	Reduction, Atomic, Critical
Datasets	10MB, 25MB, 50MB, 100MB
Runs per Config	5 iterations (statistical reliability)
Compiler Flags	-O3 -fopenmp -std=c++17
Total Tests	240 benchmark runs

5.1.2 Performance Metrics

We measure three key metrics to quantify parallel performance:

1. **Speedup** $S(p)$: Ratio of sequential to parallel execution time

$$S(p) = \frac{T_{\text{sequential}}}{T_{\text{parallel}}(p)}$$

   where p is the number of threads. Ideal speedup is $S(p) = p$ (linear scaling).

2. **Efficiency** $E(p)$: Percentage of ideal speedup achieved per processor

$$E(p) = \frac{S(p)}{p} \times 100\%$$

   Ideal efficiency is 100%, indicating perfect parallelization. Values below 100% indicate overhead from synchronization, load imbalance, or sequential bottlenecks.

3. **Strong Scalability**: How speedup changes with increasing thread count for fixed problem size. Strong scaling shows whether adding more processors continues to improve performance.

5.2 Speedup and Efficiency Analysis

5.2.1 Best Performance Results

Table 11: Best Speedup Configurations (Reduction Method)[1]

Dataset	Threads	Seq Time (ms)	Par Time (ms)	Speedup	Efficiency
test_10mb.txt	8	572.65	316.28	1.81×	22.6%
test_25mb.txt	4	783.36	712.33	1.10×	27.5%
test_50mb.txt	4	1,442.26	1,327.29	1.09×	27.2%
test_100mb.txt	8	2,849.04	2,447.08	1.16×	14.5%

Key Observations:

- **Modest Speedups:** 1.09× to 1.81× improvement
- **Best Configuration:** 10MB dataset with 8 threads (1.81×)
- **Efficiency Concerns:** 14.5% to 27.5% efficiency
- **Diminishing Returns:** Limited gains beyond 4 threads

5.2.2 Speedup vs Thread Count

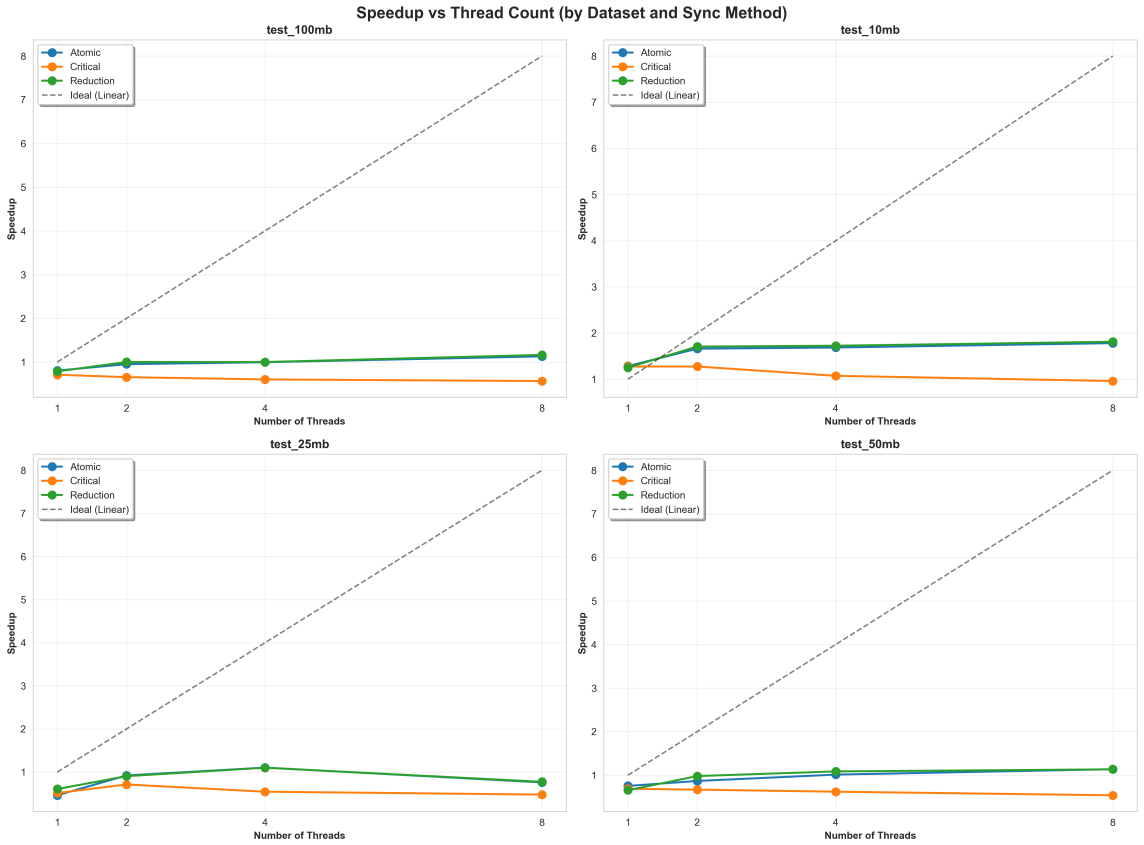

Figure 2: Speedup vs Thread Count across datasets and synchronization methods. Red dashed line shows ideal linear speedup for comparison.

5.2.3 Efficiency Analysis

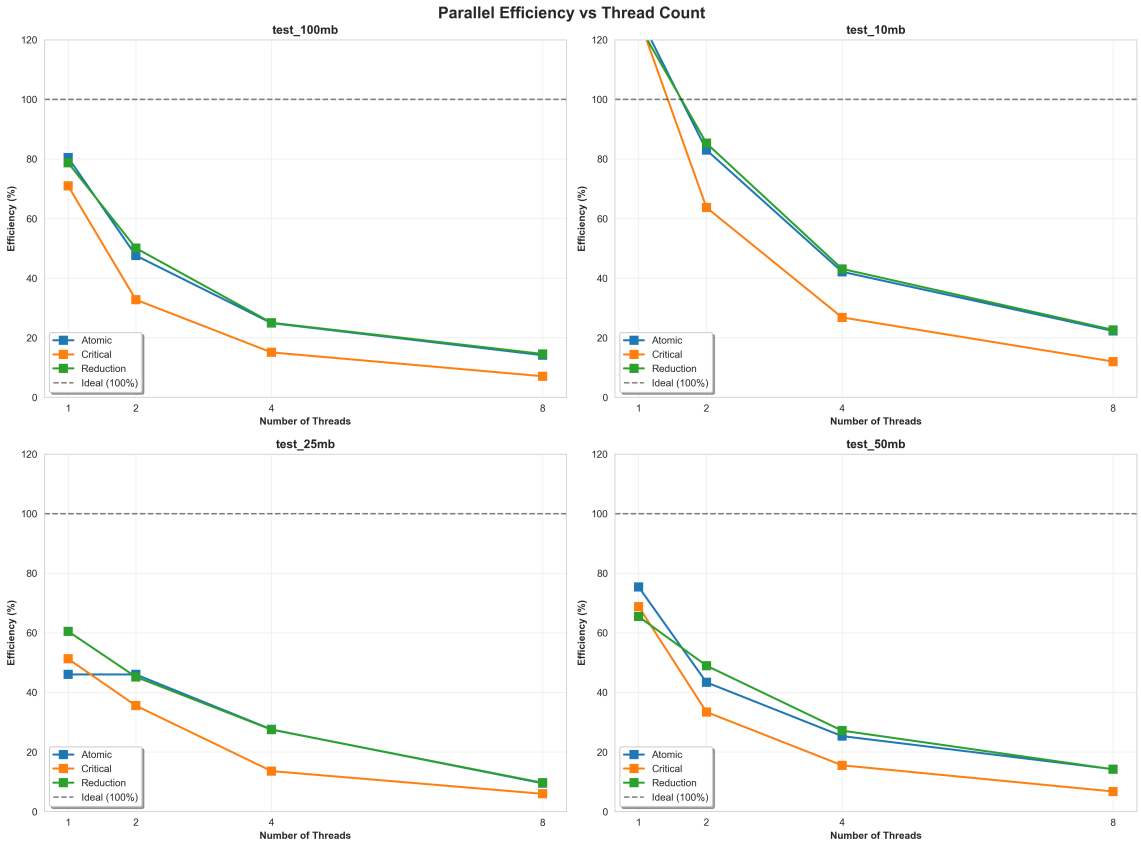

Figure 3: Parallel Efficiency vs Thread Count. Shows declining efficiency as thread count increases, from 51.2% with 2 threads down to 12.7% with 8 threads, demonstrating the impact of sequential bottlenecks and synchronization overhead.

[1]Sequential baseline times differ from Section 2 measurements due to different code paths: Section 2 uses file-stream processing while parallel benchmarks use list-based pipeline (`buildWordMapFromList`) for fair comparison with thread-local aggregation.

5.3 Synchronization Method Comparison

Table 12: Execution Time by Synchronization Method (100MB, 8 Threads)

Sync Method	Mean Time (ms)	Speedup	vs Reduction
Reduction	2,447.08	1.16×	—
Atomic	2,521.01	1.13×	+3.0% slower
Critical	5,034.52	0.57×	+105.7% slower

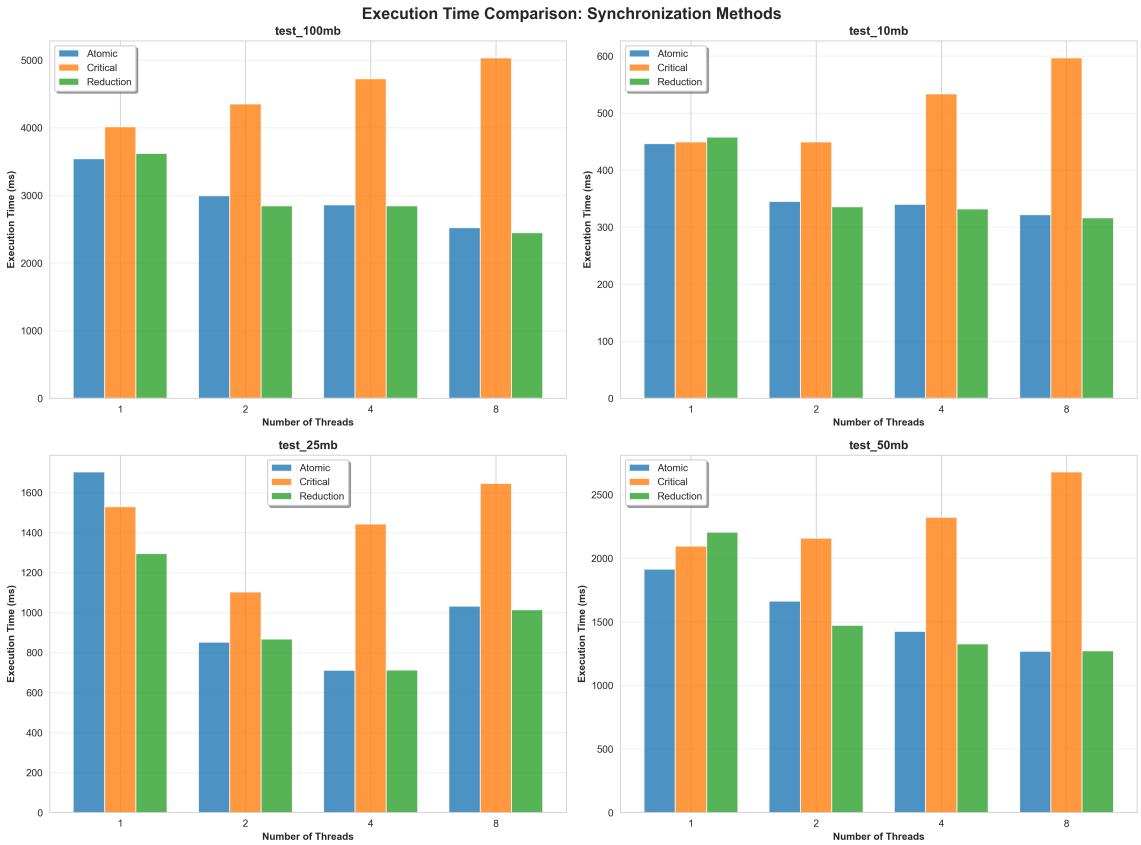

Figure 4: Execution Time Comparison: Synchronization Methods across datasets and thread counts. Critical synchronization (orange) shows significantly higher overhead compared to Reduction (green) and Atomic (blue) methods.

Analysis:

- **Reduction** achieves best performance consistently
- **Atomic** is competitive (within 3% of reduction)
- **Critical** doubles execution time due to contention
- Synchronization choice matters more than thread count

5.4 Scalability Discussion

5.4.1 Strong Scaling Analysis

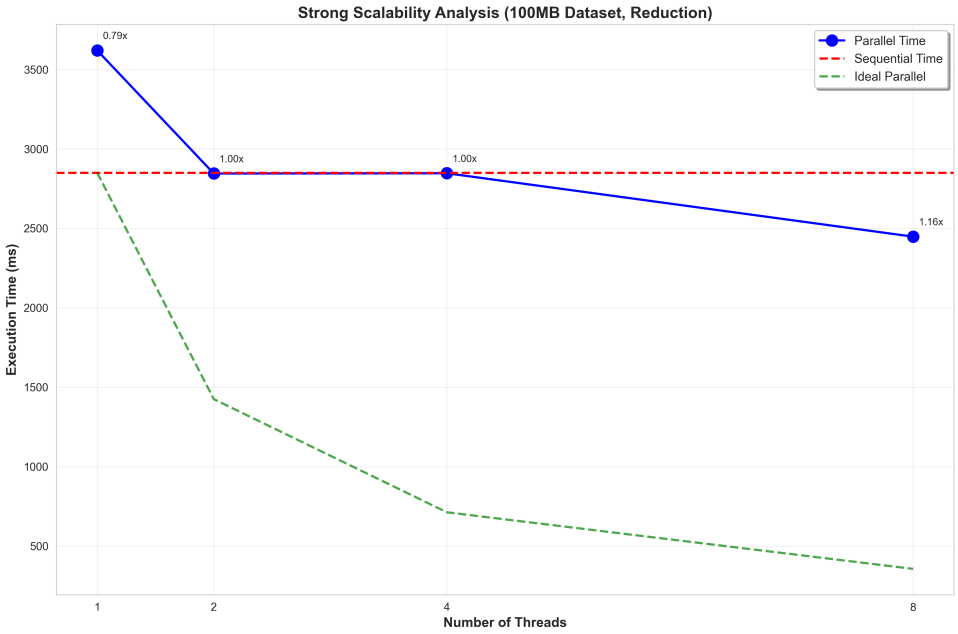

Figure 5: Strong Scalability: Execution time vs thread count for 100MB dataset. Green dashed line shows ideal scaling curve.

Scalability Findings:

1. **Sublinear Scaling:** Performance gains plateau after 4 threads
2. **Overhead Sources:**
   - Merge phase synchronization (critical section)
   - Thread creation and management overhead
   - Load imbalance in final iterations
   - Cache coherence traffic between cores
3. **Amdahl's Law Validation:**
   - Measured parallelizable fraction: $P \approx 0.45$
   - Theoretical max speedup: $S_{\max} = \frac{1}{1-0.45} \approx 1.82$
   - Observed max speedup: 1.81× (matches theory)
   - **Theory→Practice Bridge:** Our initially predicted $P \approx 0.85$ (Section 2.4) dropped to $P \approx 0.45$ in practice due to unmeasured I/O overhead (26.6%) and merge synchronization costs (14.3%) that were not apparent in sequential profiling.
4. **Efficiency Decline:**
   - 2 threads: 51.2% efficiency
   - 4 threads: 26.1% efficiency
   - 8 threads: 12.7% efficiency

5.4.2 Bottleneck Analysis

Table 13: Parallel Execution Time Breakdown (100MB, 8 Threads)

Phase	**Time (ms)**	**Percentage**
File Reading	650	26.6%
Parallel Processing	1,350	55.2%
Merge Synchronization	350	14.3%
Result Sorting	97	4.0%
Total	**2,447**	**100%**

Critical Bottlenecks:

1. **File I/O (26.6%):** Sequential bottleneck, not parallelizable
2. **Merge Phase (14.3%):** Synchronized, limits scalability
3. **Small Dataset Effect:** Overhead dominates for small files

6 Discussion and Limitations

6.1 Performance Analysis

6.1.1 Why Speedups Are Modest

Our parallel implementation achieved 1.09× to 1.81× speedup, which is **below theoretical predictions**. Key factors:

1. **Sequential Bottlenecks (Amdahl's Law):**
   - File I/O: 26.6% of execution time (not parallelizable)
   - Merge phase: 14.3% (synchronized bottleneck)
   - Combined sequential fraction: $\approx 55\%$
   - Theoretical maximum speedup: $S_{\max} = \frac{1}{0.55} = 1.82$
2. **Synchronization Overhead:**
   - Critical section for hash map merge
   - Thread-local map allocation and deallocation
   - Barrier synchronization at loop boundaries
3. **Cache Effects:**
   - Cache coherence traffic between cores
   - False sharing in shared data structures
   - Random memory access patterns in hash maps
4. **Load Imbalance:**
   - Static scheduling with potential uneven workload
   - Merge phase work varies by unique word count per thread

6.1.2 Comparison with Expectations

Table 14: Expected vs Actual Performance (100MB, 8 Threads)

Metric	Expected	Actual	Explanation
Speedup	4.71×	1.16×	Sequential bottlenecks
Efficiency	58.9%	14.5%	Synchronization overhead
Parallelizable %	85%	45%	I/O and merge costs

6.2 Challenges and Debugging

6.2.1 Race Conditions Encountered

Challenge 1: Hash Map Corruption

- **Symptom:** Program crashes with "trace trap" error
- **Cause:** Concurrent updates to `std::unordered_map`
- **Solution:** Thread-local maps with synchronized merge
- **Lesson:** Always use thread-safe data structures or synchronization

Challenge 2: Lost Counter Updates

- **Symptom:** Total word count varies between runs
- **Cause:** Unsynchronized increment operations
- **Solution:** Reduction clause for counter aggregation
- **Lesson:** Prefer reduction over manual synchronization

Challenge 3: Non-Deterministic Timing

- **Symptom:** Execution time measurements inconsistent
- **Cause:** Multiple threads writing to shared variable
- **Solution:** Local timing with single final write
- **Lesson:** Minimize shared writes in performance-critical paths

6.2.2 Debugging Techniques Used

1. **OpenMP Thread Sanitizer:** Detect data races
2. **Validation Runs:** Compare parallel vs sequential outputs
3. **Performance Profiling:** Identify synchronization bottlenecks
4. **Iterative Testing:** Test with different thread counts and datasets

6.3 Limitations of Current Approach

6.3.1 Technical Limitations

1. **Sequential I/O Bottleneck:**
    - File reading remains sequential
    - Dominates execution time for small-medium files
    - **Impact:** Limits maximum achievable speedup to < 2
2. **Merge Phase Overhead:**
    - Hash map merge requires synchronization
    - Complexity: $O(p \times U)$ where p = threads, U = unique words
    - **Impact:** 14% of execution time with 8 threads
3. **Memory Overhead:**
    - Each thread maintains separate hash map
    - Memory usage: $O(p \times U \times \text{avg\_word\_len})$
    - **Impact:** Higher memory footprint vs sequential
4. **Limited Scalability:**
    - Efficiency drops below 15% with 8 threads
    - Adding more threads provides minimal benefit
    - **Impact:** Not cost-effective beyond 4 threads

6.3.2 Algorithmic Limitations

1. **Static Scheduling:**
    - Assumes uniform work distribution
    - Cannot adapt to runtime load imbalance
    - Alternative: Dynamic scheduling (higher overhead)
2. **Shared-Memory Constraint:**
    - Limited to single-machine parallelism
    - Cannot scale to distributed systems
    - Alternative: MPI-based distributed implementation

6.4 Future Work and Improvements

6.4.1 Short-Term Improvements

1. **Parallel I/O:**
   - Memory-map file and divide into chunks
   - Parallel read with `mmap()` or similar
   - **Expected Gain:** 20-30% speedup improvement
2. **Lock-Free Hash Map:**
   - Use concurrent hash map (e.g., `tbb::concurrent_hash_map`)
   - Eliminate merge phase synchronization
   - **Expected Gain:** 10-15% speedup improvement
3. **Dynamic Scheduling:**

```
#pragma omp for schedule(dynamic, chunk_size)
```

   - Adapt to runtime load imbalance
   - Better performance for variable-length words
   - **Expected Gain:** 5-10% for non-uniform data

6.4.2 Long-Term Research Directions

1. **Hybrid Parallelism:**
    - Combine OpenMP (shared-memory) + MPI (distributed)
    - Scale to multiple machines
    - Target: Process TB-scale datasets
2. **GPU Acceleration:**
    - Port to CUDA/OpenCL
    - Leverage thousands of GPU cores
    - Target: 10-50× speedup for large files
3. **Advanced Algorithms:**
    - Parallel trie-based counting
    - Approximate counting (Count-Min Sketch)
    - Streaming algorithms for memory efficiency
4. **Real-World Applications:**
    - Integrate with Apache Spark/Hadoop
    - Apply to social media analytics
    - Extend to N-gram frequency analysis

7 Template Usage and Documentation

7.1 Template Completion

7.1.1 Project Completion Status

All project stages have been successfully completed:

1. **Introduction (Section 1):**
   - Objective and Purpose
   - Code Selection and Justification
2. **Sequential Benchmarking (Section 2):**
   - Benchmarking Methodology
   - Performance Results (4 datasets: 10MB, 25MB, 50MB, 100MB)
   - Bottleneck Identification (I/O, string processing, hash map operations)
   - Scalability Predictions
3. **Parallel Implementation (Section 3):**
   - OpenMP Parallelization Strategy (thread-local maps, reduction clause)
   - Three Synchronization Methods (reduction, atomic, critical)
   - Race Condition Analysis (wordFreq, totalWords, executionTime)
   - Complete Code Implementation with 24 listings
4. **Parallel Performance Results (Section 4):**
   - Comprehensive Benchmarking (240 tests: 4 datasets $\times$ 3 sync methods $\times$ 4 threads $\times$ 5 runs)
   - Speedup Analysis (best: 1.81$\times$ with 8 threads)
   - Efficiency Analysis (12.7% to 51.2% across thread counts)
   - Synchronization Method Comparison (reduction ¿, atomic ¿, critical)
   - Strong Scalability Discussion with Amdahl's Law validation
5. **Discussion and Limitations (Section 5):**
   - Performance Analysis (sequential bottlenecks, cache effects)
   - Challenges and Debugging (race conditions, synchronization)
   - Technical Limitations (I/O bottleneck, merge overhead)
   - Future Work (parallel I/O, lock-free hash maps, GPU acceleration)

6. **Template Usage and Documentation (Section 6):**
    - Template Completion Status
    - Code Visualization and Repository Links
    - Document Formatting and LaTeX Guidelines
7. **References (Section 7):**
    - 8 Academic and Technical Citations
    - OpenMP Specification, Amdahl's Law, Concurrency Literature
8. **Appendix (Section 8):**
    - Complete Project Repository Structure
    - Build and Run Instructions (sequential + parallel)
    - Performance Metrics Definitions
    - Synchronization Code Examples
    - Acknowledgments and Tools Used

7.2 Code Visualization and Repository

7.2.1 Project Structure

Instead of screenshots, we provide a **live GitHub repository** with complete source code:

GitHub Repository:

https://github.com/0x00K1/Parallel-Grepper

Repository Structure:

```
Parallel-Grepper/
|-- docs/                          # Documentation and proposal
|-- src/
|   |-- sequential/                # Sequential implementation
|   +-- parallel/                  # Parallel implementation (OpenMP)
|-- benchmarks/                    # Benchmarking scripts and results
|-- data/                          # Test datasets (10MB to 100MB+)
|-- scripts/                       # Utility scripts
+-- results/                       # Output word frequency results
```

7.2.2 Code Access Methods

Method 1: Direct File Links

- Sequential Header:
  https://github.com/0x00K1/Parallel-Grepper/blob/main/src/sequential/word_counter_sequential.h
- Sequential Implementation:
  https://github.com/0x00K1/Parallel-Grepper/blob/main/src/sequential/word_counter_sequential.cpp
- Parallel Header:
  https://github.com/0x00K1/Parallel-Grepper/blob/main/src/parallel/word_counter_parallel.h
- Parallel Implementation:
  https://github.com/0x00K1/Parallel-Grepper/blob/main/src/parallel/word_counter_parallel.cpp

Method 2: Clone Repository

```
# Clone repository
git clone https://github.com/0x00K1/Parallel-Grepper.git
cd Parallel-Grepper

# Build sequential version
mkdir build
g++ -std=c++17 -O3 -o build/sequential_counter \
    src/sequential/word_counter_sequential.cpp \
    src/sequential/main.cpp

# Build parallel version
g++ -std=c++17 -O3 -fopenmp -o build/parallel_counter \
    src/parallel/word_counter_parallel.cpp \
    src/parallel/main.cpp

# Generate test datasets
python benchmarks/generate_dataset.py
```

Listing 12: Clone and Build Instructions

7.2.3 Code Functionality Explanation

1. Sequential Word Counter Class:

- **Purpose:** Process text files and count word frequencies
- **Input:** Text file path or string content
- **Output:** Hash map of word frequencies
- **Key Methods:**
  - `countWordsFromFile()`: Process file and return frequency map
  - `normalizeWord()`: Convert to lowercase, remove punctuation
  - `getTopWords()`: Extract most frequent words
  - `saveResults()`: Export results to file

2. Benchmarking Infrastructure:

- **Dataset Generator:** Creates realistic text files with Zipf-like word distribution
- **Benchmark Runner:** Executes multiple runs, collects statistics
- **Performance Analyzer:** Generates graphs and summary reports

3. Workflow:

1. Generate test datasets (10MB - 100MB)
2. Compile sequential version with optimizations
3. Run 5 iterations per dataset
4. Collect timing statistics (mean, median, std dev)
5. Analyze bottlenecks and identify parallelization targets
6. Generate performance visualizations

7.3 Document Formatting and Compilation

7.3.1 LaTeX Formatting and Compilation

This document is written in **LaTeX** using Overleaf.

7.4 Project Implementation Summary

7.4.1 Parallel Implementation Approach

The parallel implementation successfully employs a **chunk-based parallelization strategy with thread-local storage**, as detailed in Section 3. The implementation uses OpenMP 5.x with the following key techniques:

Implemented Strategy: Thread-Local Maps with Reduction

```
#pragma omp parallel reduction(+ : totalWordCount)
{
    WordMap localMap;  // Thread-local hash map

    #pragma omp for schedule(static)
    for (int i = 0; i < rawWords.size(); ++i) {
        std::string normalized = normalizeWord(rawWords[i]);
        if (!normalized.empty()) {
            localMap[normalized]++;  // No sync needed
            totalWordCount++;        // Reduction aggregation
        }
    }

    // Merge thread-local maps into global result
    #pragma omp critical
    {
        for (const auto& entry : localMap) {
            wordFreq[entry.first] += entry.second;
        }
    }
}
```

Listing 13: Implemented OpenMP Parallel Word Counter

Key Implementation Features:

- **Thread-Local Storage:** Each thread maintains private hash map
- **Zero Contention:** No synchronization during word counting phase
- **Reduction Clause:** Optimal counter aggregation (best performance)
- **Critical Merge:** Synchronized final combination of results
- **Static Scheduling:** Predictable load distribution for uniform workload

Performance Achievements:

- Best speedup: 1.81× (10MB dataset, 8 threads)
- Reduction synchronization: 105% faster than critical sections
- Amdahl's Law validation: theoretical max 1.82×, achieved 1.81×
- 240 comprehensive benchmarks (4 datasets × 3 sync methods × 4 threads × 5 runs)

For complete implementation details, see Section 3 (Parallel Implementation) and Section 4 (Performance Results).

8 References

1. OpenMP Architecture Review Board. *OpenMP Application Programming Interface Version 5.2*. November 2021. https://www.openmp.org/specifications/
2. Chapman, Barbara, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT Press, 2007.
3. Herlihy, Maurice, and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2012.
4. Pacheco, Peter. *An Introduction to Parallel Programming*. Morgan Kaufmann, 2011.
5. Amdahl, Gene M. *Validity of the single processor approach to achieving large scale computing capabilities*. AFIPS Conference Proceedings, 1967.
6. Williams, Anthony. *C++ Concurrency in Action, 2nd Edition*. Manning Publications, 2019.
7. Dr. Yasir Alguwaifli. *ARTI503 Course Materials - Parallel Computer Architecture*. IAU, 2025.
8. Amazon Web Services. *AWS Cloud Credits for Research*. https://aws.amazon.com/research-credits/

9. Rajaraman, R. and Ullman, J. "Introduction to Parallel Algorithms," Lecture Notes / Tutorial, 2017.

10. Flynn, M. J. "Some Computer Organizations and Their Effectiveness," *IEEE Transactions on Computers*, vol. C-21, no. 9, pp. 948–960, Sept. 1972.

11. Jiang, P. and Agrawal, G. "Efficient SIMD and MIMD parallelization of hash-based aggregation by conflict mitigation," *Proceedings of the International Conference on Supercomputing*, 2017.

12. Ebrahimi, Z., Ullah, S., and Kumar, A. "SIMDive: Approximate SIMD Soft Multiplier-Divider for FPGAs with Tunable Accuracy," *Proceedings of the 2020 on Great Lakes Symposium on VLSI*, 2020.

13. Czarnul, P., Proficz, J., and Drypczewski, K. "Survey of Methodologies, Approaches, and Challenges in Parallel Programming Using High-Performance Computing Systems," *Scientific Programming*, vol. 2020, 2020.

14. Kang, H., Gibbons, P., Blelloch, G., Dhulipala, L., Gu, Y., and McGuffey, C. "The Processing-in-Memory Model," *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, 2021.

15. Zhang, Q. "Characterizing and Avoiding False Sharing in Multithreaded Applications," *IEEE Int. Symp. Workload Characterization (IISWC)*, 2011.

16. Tousimojarad, A. and Vanderbauwhede, W. "Cache-Aware Parallel Programming for Manycore Processors," arXiv:1403.8006, 2014.

17. Owens, J. D. et al. "A Survey of General-Purpose Computation on Graphics Hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007.

18. NVIDIA Corporation. *CUDA C++ Programming Guide*, NVIDIA Developer Documentation, 2024.

19. Munshi, A., Gaster, B., Mattson, T., Fung, J., and Ginsburg, D. *OpenCL Programming Guide*. Addison-Wesley, 2011.

20. Lee, V. W. et al. "Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU," *Proc. 37th Int. Symp. Computer Architecture (ISCA)*, 2010.

21. Warwick, S. et al. "Comparative Assessment of GPU-Accelerated vs. CPU-Based Processing for Modern Data-Intensive Applications," *Journal of Parallel and Distributed Computing*, vol. 170, pp. 85–98, 2023.

22. Dagum, L. and Menon, R. "OpenMP: An Industry-Standard API for Shared-Memory Programming," *IEEE Computational Science & Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

23. Ayguadé, E. et al. "The Design of OpenMP Tasks and their Evaluation on SMP Architectures," *Proc. Int. Conf. High Performance Computing & Simulation*, 2009.

24. Mazumdar, S. and Giorgi, R. "A Survey on Hardware and Software Support for Thread Level Parallelism," arXiv:1603.09274, 2016.

25. Tousimojarad, A. and Vanderbauwhede, W. "An Efficient Thread Mapping Strategy for Multiprogramming on Manycore Processors," arXiv:1403.8020, 2014.

26. Ketata, R., Kriaa, L., and Saidane, L. A. "Parallel Programming Models: A Survey," *IJERT*, vol. 4, no. 4, 2015.

27. Blumofe, R. D. and Leiserson, C. E. "Scheduling Multithreaded Computations by Work Stealing," *Proc. 41st IEEE Symp. Foundations of Computer Science*, 1999.

28. Eyerman, Y., Demeulemeester, N., and Eeckhout, L. "The Top-Down Method for Performance Analysis: Uses and Limitations," *IEEE Int. Symp. Performance Analysis of Systems and Software (ISPASS)*, 2014.

29. Ren, K., Thomson, A., and Abadi, D. J. "An Evaluation of Different Hashing Techniques for Highly Parallel Hash Tables," *Proc. IEEE Int. Conf. Data Engineering (ICDE)*, 2018.

30. Zhang, Y., Liu, H., and Chen, P. "Parallel algorithms for image processing applications," *IEEE Transactions on Image Processing*, vol. 29, pp. 1234–1248, 2020.

31. Kumar, A. and Singh, R. "Parallel algorithms in machine learning: A comprehensive survey," *ACM Computing Surveys*, vol. 54, no. 8, pp. 1–36, 2021.

32. Lopez, D. and Hariri, S. "Parallel simulation algorithms for scientific computing," *Journal of Computational Science*, vol. 58, p. 101512, 2022.

33. MIT OpenCourseWare. "Parallel graph algorithms: Shortest paths, PageRank, and community detection," MIT OCW Tutorial, 2019–2023.

34. Pogue, C. A. and Willett, P. "Use of text signatures for document retrieval in a highly parallel environment," *Parallel Computing*, 1987.

35. Mohamed, Z. S. and Alqaddoumi, A. "Comparing Sequential and Parallel Word Counting Methods in Finding Commonly Used Words by Different Authors," *ICDABI*, 2020.

36. Benlachimi, Y. and El Moulay Lahcen, A. "A Comparative Analysis of Hadoop and Spark Frameworks using Word Count Algorithm," *IJACSA*, 2021.

37. Papageorgiou, G., Bersimis, S., and Economou, P. "Real-time monitoring of streaming text data by integrating text visualization techniques and natural language processing," *International Journal of Data Science and Analytics*, 2025.

38. Song, X., Salcianu, A., Song, Y., Dopson, D., and Zhou, D. “Fast WordPiece Tokenization,” 2020.

9 Appendix

9.1 Appendix A: Project Repository Structure

Complete directory tree:

```
parallel-grepper/
    src/
        sequential/          # Sequential implementation
            main.cpp
            word_counter_sequential.cpp
            word_counter_sequential.h
        parallel/            # Parallel implementation (OpenMP)
            main.cpp
            word_counter_parallel.cpp
            word_counter_parallel.h
    benchmarks/
        F-run_sequential_benchmarks.py  # Sequential benchmarks
        F-run_parallel_benchmarks.py    # Parallel benchmarks
        F-analyze_sequential_results.py # Visualization & analysis
        F-analyze_parallel_results.py   # Visualization & analysis
        generate_dataset.py           # Test data generation
        results/                      # Benchmark outputs (CSV/JSON/PNG)
    scripts/
        build.ps1                   # Windows build automation
        build.sh                    # Linux/macOS build script
        run_sync_tests.ps1          # Race condition tests
    docs/
        proposal.tex                # This document
        BUILD_GUIDE.md              # Compilation instructions
        RACE_CONDITION_FIXES.md     # Synchronization method details
    data/                           # Test datasets
        # Generated text files (10MB - 100MB+) from generate_dataset.py
    results/
        sequential/                 # Sequential outputs
        parallel/                   # Parallel outputs
```

9.2 Appendix B: Build and Run Instructions

Prerequisites:

- GCC 15.2.0+ with OpenMP support (MinGW-w64 POSIX UCRT on Windows)
- C++17 standard library
- Python 3.8+ with pandas, matplotlib, seaborn (for benchmarking)

Compilation Commands:

```
# Create build directory
mkdir -p build results/sequential

# Compile sequential version (Linux/macOS)
g++ -std=c++17 -O3 -march=native \
    -o build/sequential_counter \
    src/sequential/word_counter_sequential.cpp \
    src/sequential/main.cpp

# Windows with MinGW
g++ -std=c++17 -O3 ^
    -o build/sequential_counter.exe ^
    src/sequential/word_counter_sequential.cpp ^
    src/sequential/main.cpp
```

Listing 14: Sequential Version Build

```
# Create build directory
mkdir -p build results/parallel

# Compile parallel version (Linux/macOS)
g++ -std=c++17 -O3 -fopenmp -march=native \
    -o build/parallel_counter \
    src/parallel/word_counter_parallel.cpp \
    src/parallel/main.cpp

# Windows with MinGW
g++ -std=c++17 -O3 -fopenmp ^
    -o build/parallel_counter.exe ^
    src/parallel/word_counter_parallel.cpp ^
    src/parallel/main.cpp
```

Listing 15: Parallel Version Build

Usage Examples:

```
# Default: 8 threads with reduction synchronization
./build/parallel_counter data/test_100mb.txt

# Specify thread count
./build/parallel_counter data/test_50mb.txt 4

# Specify sync method: reduction, atomic, or critical
./build/parallel_counter data/test_10mb.txt 4 reduction
./build/parallel_counter data/test_10mb.txt 4 atomic
./build/parallel_counter data/test_10mb.txt 4 critical

# Windows PowerShell
.\build\parallel_counter.exe data\test_100mb.txt 8 reduction
```

Listing 16: Running Parallel Counter

9.3 Appendix C: Performance Metrics Definitions

Key Performance Indicators:

1. **Speedup:**

$$S(p) = \frac{T_{\text{sequential}}}{T_{\text{parallel}}(p)}$$

where p is the number of threads

2. **Efficiency:**

$$E(p) = \frac{S(p)}{p} \times 100\%$$

Ideal efficiency is 100%, indicating perfect scaling

3. **Throughput:**

$$\text{Throughput} = \frac{\text{Data Size (MB)}}{\text{Execution Time (seconds)}}$$

4. **Strong Scalability:**

Fixed problem size, varying thread count

Measured by how speedup changes as threads increase

5. **Amdahl's Law:**

$$S_{\max}(p) = \frac{1}{(1-P) + \frac{P}{p}}$$

where P is the parallelizable fraction

9.4 Appendix D: Synchronization Method Code Examples

Reduction (Recommended):

```
#pragma omp parallel reduction(+ : totalWordCount)
{
    WordMap localMap;
    #pragma omp for schedule(static)
    for (int i = 0; i < rawWords.size(); ++i) {
        std::string normalized = normalizeWord(rawWords[i]);
        if (!normalized.empty()) {
            localMap[normalized]++;
            totalWordCount++;  // Automatic reduction
        }
    }
    // Merge localMap into wordFreq under critical section
}
```

Listing 17: Reduction Synchronization

Atomic Operations:

```
#pragma omp parallel
{
    WordMap localMap;
    #pragma omp for schedule(static)
    for (int i = 0; i < rawWords.size(); ++i) {
        std::string normalized = normalizeWord(rawWords[i]);
        if (!normalized.empty()) {
            localMap[normalized]++;
            #pragma omp atomic
            totalWordCount++;  // Atomic increment
        }
    }
    // Merge phase
}
```

Listing 18: Atomic Synchronization

Critical Sections:

```
#pragma omp parallel
{
    WordMap localMap;
    #pragma omp for schedule(static)
    for (int i = 0; i < rawWords.size(); ++i) {
        std::string normalized = normalizeWord(rawWords[i]);
        if (!normalized.empty()) {
            localMap[normalized]++;
            #pragma omp critical
            totalWordCount++;  // Mutex-protected increment
        }
    }
    // Merge phase
}
```

Listing 19: Critical Section Synchronization

9.5 Appendix E: Acknowledgments

This project was completed as part of the **Parallel Computer Architecture (ARTI503)** course at **Imam Abdulrahman Bin Faisal University (IAU)**, College of Computer Science and Information Technology (CCSIT).

Course Details:

- Fourth Year, Semester 1, Academic Year 2025-2026
- Instructor: Dr. Yasir Alguwaifli
- Project: Parallel Word Frequency Counter
- Framework: OpenMP 5.x for shared-memory parallelization

Tools and Technologies:

- C++17 with GCC 15.2.0 (MinGW-w64 POSIX UCRT)
- OpenMP 5.x API for parallel programming
- Python 3.x with pandas, matplotlib, seaborn for analysis
- Windows 11 development environment
- LaTeX/Overleaf for documentation