SISMD

ISEP – Mestrado em Engenharia Informática

M1A

Authors

Martim Oliveira - 1181754

Rafael Gomes – 1211426

David Ferreira – 1240444

Porto, 12 de Maio 2025

Index

Introduction

This project was developed as part of the Sistemas Multinúcleo e Distribuídos (SISMD) course and focuses on exploring and analysing different parallel and concurrent programming techniques for efficiently processing large volumes of data on multicore systems.

This study implements and analyzes five distinct approaches to process a large Wikipedia data dump, measuring the frequency of words in English texts, by leveraging the parallel processing capabilities of modern multicore architectures, we aim to demonstrate significant performance improvements over traditional sequential methods. The implementations range from basic multithreading to sophisticated asynchronous programming models, each with distinct characteristics and performance implications.

The core task is counting word frequencies in large text corpora, this represents a common pattern in data processing applications, while simple, this problem becomes computationally intensive at scale, making it an ideal candidate for parallelization. Our analysis focuses not only on raw performance gains but also on the scalability, resource utilization, overhead and bottlenecks.

The project is structured into several sections, each detailing a specific implementation approach, its design considerations, concurrency and synchronization, the different metrics gathered and finally the conclusion.

Objectives

This project aims to implement and thoroughly analyse different approaches to efficiently process large volumes of textual data on multicore systems, by focusing on the task of word frequency analysis in a Wikipedia data dump, we seek to evaluate the performance characteristics of various concurrency models, identify their strengths and limitations, and compare them.

The project's objectives are divided into two main categories: implementation and analysis. Implementation objectives focus on developing and optimizing different concurrent solutions, while analysis objectives emphasize measuring performance, resource utilization, and scalability.

Implementation Objectives

1. **Develop a baseline sequential solution** that processes the Wikipedia dataset without any parallelization, serving as a reference point for performance comparisons.
2. **Implement an explicit multithreaded solution** with manually managed threads, focusing on effective work distribution and proper synchronization mechanisms.
3. **Create a thread pool-based implementation** that efficiently manages thread lifecycles, reducing the overhead associated with thread creation and destruction.
4. **Design a solution using the Fork/Join framework** to utilize recursive task decomposition and work-stealing algorithms for balanced workload distribution.
5. **Develop an asynchronous implementation using CompletableFutures** to handle task dependencies and composition without explicit thread management.
6. **Apply and evaluate garbage collector tuning strategies** to optimize memory management and improve overall application performance.

Analysis Objectives

1. **Measure and compare execution times** across all implementations to quantify performance improvements over the sequential baseline.
2. **Analyze resource utilization patterns** including CPU usage, memory consumption, and thread activity during execution.

3. **Evaluate scalability characteristics** by testing each implementation with varying dataset sizes and available processor cores.
4. **Identify performance bottlenecks and synchronization overhead** in each concurrent approach.
5. **Determine optimal concurrency strategies** for word frequency analysis based on different system configurations and dataset characteristics.

Implementation

Sequential Implementation

Overview

The sequential implementation processes Wikipedia pages one-by-one, counting word occurrences using a single thread.

Core Components

Sequential Processing Loop

```
for(Page page: pages) {
 if(page == null) break;
 Iterable<String> words = new Words(page.getText());
 for (String word: words)
   if(word.length()>1 || word.equals("a") || word.equals("I"))
     countWord(word);
 ++processedPages;
}
```

This loop iterates through each Wikipedia page, extracts words, and counts them sequentially.

Word Counting

```
 private static void countWord(String word) {
  counts.merge(word, 1, Integer::sum);
}
```

The merge method efficiently updates the count by either inserting a new word with count 1 or incrementing an existing word's count.

Results Processing

```
LinkedHashMap<String, Integer> commonWords = new LinkedHashMap<>();
counts.entrySet().stream()
      .sorted(Map.Entry.comparingByValue(Comparator.reverseOrder()))
      .forEachOrdered(x -> commonWords.put(x.getKey(), x.getValue()));
```

This section sorts words by frequency and stores them in a LinkedHashMap to preserve order.

Multithreaded(No Thread Pool) Implementation

Overview

This implementation uses multithreading to efficiently process a large Wikipedia XML dump file (`enwiki.xml`), it follows a **producer-consumer** model where a single thread reads pages, and multiple threads process them in parallel to count word occurrences.

Instead of sequentially processing pages, this implementation uses a **blocking queue** to decouple the reading (producer) and processing (consumers) tasks. This enables parallel execution and improves throughput on multi-core systems.

Core Components

Producer Thread

```
Thread producer = new Thread(() -> {
    Iterable<Page> pages = new Pages(MAX_PAGES, FILE_NAME);
    try {
        for (Page page : pages) {
            if (page == null) continue;
            queue.put(page);
            System.out.println("Producer added page: " + page.getTitle())
;
        }
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    } finally {
        producerDone.set(true);
    }
});
```

- Reads pages from the XML file.

- Puts each valid page into the blocking queue.
- Sets `producerDone = true` once finished.

Consumer Threads

```
Thread consumer = new Thread(() -> {
    try {
        while (true) {
            if (producerDone.get() && queue.isEmpty()) break;

            Page page = queue.poll(100, TimeUnit.MILLISECONDS);
            if (page != null) {
                System.out.println(Thread.currentThread().getName() + " p
rocessing page: " + page.getTitle());
                mapReduce.map(page.getText());
            }
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
});
```

- Continuously polls the queue.
- Processes page text with `mapReduce.map()` if available.
- Exits when producer is done and the queue is empty.

Word Counting Logic (MapReduce)

```
public void map(String text) {
    for (String word : text.split("\\W+")) {
        if (!word.isEmpty()) {
            String lower = word.toLowerCase();
            wordCounts.merge(lower, 1, Integer::sum);
        }
    }
}
```

- Splits text using non-word characters.
- Converts to lowercase.
- Uses `ConcurrentHashMap.merge()` for thread-safe counting.

Results Aggregation

```
public void printTopWords(int topN) {
    wordCounts.entrySet().stream()
        .sorted((a, b) -> b.getValue().compareTo(a.getValue()))
        .limit(topN)
        .forEach(entry ->
            System.out.println("Word: '" + entry.getKey() + "' occurred "
```

```
+ entry.getValue() + " times!"));
}
```

- Sorts all words by frequency.
- Displays the top N most frequent words.

Multithreaded(Thread Pool) Implementation

Overview

This implementation processes a large number of Wikipedia pages using Java's **ExecutorService** and a fixed thread pool to perform concurrent word counting.

Instead of manually managing threads, this version uses a **thread pool** via `Executors.newFixedThreadPool()`, each task submitted to the pool processes a Wikipedia page, extracting and counting words in parallel.

Core Components

Executor Service Setup

```
ExecutorService executor = Executors.newFixedThreadPool(NUM_THREADS);
```

- Creates a fixed-size thread pool to manage worker threads.
- Allows concurrent execution of submitted tasks without manually managing thread lifecycle.

Page Processing Loop

```
Iterable<Page> pages = new Pages(MAX_PAGES, FILE_NAME);

for (Page page : pages) {
    if (page == null) continue;
    executor.execute(() -> mapReduce.map(page.getText()));
}
```

- Iterates through all the pages from the file.
- Each page is submitted to the executor for parallel processing.

- `mapReduce.map()` is the method that extracts and counts words in a thread-safe way.

Graceful Shutdown

```
executor.shutdown();
try {
    if (!executor.awaitTermination(1, TimeUnit.HOURS)) {
        executor.shutdownNow();
    }
} catch (InterruptedException e) {
    executor.shutdownNow();
    Thread.currentThread().interrupt();
}
```

- Signals the executor to stop accepting new tasks.
- Waits up to 1 hour for all tasks to finish.
- Ensures clean shutdown even if interrupted.

Fork/Join Implementation

Overview

The Fork/Join implementation uses the `java.util.concurrent.ForkJoinPool` framework, which is ideal for tasks that can be **recursively divided into smaller, independent subtasks**. In this project, a recursive task (`WordCountRecursiveTask`) splits the list of Wikipedia pages and processes word counting in parallel, leveraging **work-stealing** to maximize CPU core utilization.

Core Components

Recursive Task Class

```
public class WordCountRecursiveTask extends RecursiveTask<Map<String, Int
eger>> {
    private final List<Page> pages;
    private final int start, end;
    private static final int THRESHOLD = 100;

    public WordCountRecursiveTask(List<Page> pages, int start, int end) {
        this.pages = pages;
        this.start = start;
        this.end = end;
    }
```

```
    @Override
    protected Map<String, Integer> compute() {
        if (end - start <= THRESHOLD) {
            return countWordsSequentially();
        } else {
            int mid = (start + end) / 2;
            var left = new WordCountRecursiveTask(pages, start, mid);
            var right = new WordCountRecursiveTask(pages, mid, end);
            left.fork();
            Map<String, Integer> rightResult = right.compute();
            Map<String, Integer> leftResult = left.join();
            return mergeMaps(leftResult, rightResult);
        }
    }

    private Map<String, Integer> countWordsSequentially() {
        Map<String, Integer> result = new HashMap<>();
        for (int i = start; i < end; i++) {
            Page page = pages.get(i);
            for (String word : new Words(page.getText())) {
                result.merge(word.toLowerCase(), 1, Integer::sum);
            }
        }
        return result;
    }

    private Map<String, Integer> mergeMaps(Map<String, Integer> a, Map<St
ring, Integer> b) {
        b.forEach((k, v) -> a.merge(k, v, Integer::sum));
        return a;
    }
}
```

ForkJoinPool Execution

```
ForkJoinPool pool = new ForkJoinPool();
List<Page> allPages = StreamSupport.stream(new Pages(MAX_PAGES, FILE_NAME
).spliterator(), false)
        .collect(Collectors.toList());
WordCountRecursiveTask task = new WordCountRecursiveTask(allPages, 0, all
Pages.size());
Map<String, Integer> result = pool.invoke(task);
```

Performance Observations

- The Fork/Join approach resulted in efficient workload distribution across threads, with notable performance gains on multi-core systems.
- The internal work-stealing algorithm allowed idle threads to dynamically "steal" tasks, improving parallel execution.
- Compared to the thread pool implementation, the Fork/Join version showed better scalability on larger datasets.

Considerations

- The THRESHOLD value directly affects performance:
- Low thresholds create many small tasks, increasing overhead.
- High thresholds reduce parallelism and may underutilize cores.

CompletableFuture Implementation

Overview

This implementation processes Wikipedia pages using Java's CompletableFuture API to perform concurrent word counting on a large XML corpus, the code uses CompletableFuture to asynchronously process multiple Wikipedia pages, leveraging parallelism without manually managing threads.

Core Components

Executor Service Setup

```
ExecutorService executor = Executors.newFixedThreadPool(THREAD_POOL_SIZE)
;
```

- Creates a fixed-size thread pool to manage worker threads.
- Provides the execution environment for the CompletableFuture tasks.

Asynchronous Page Processing

```
List<CompletableFuture<Map<String,Integer>>> futures = new ArrayList<>();

for (Page page : pages) {
    if (page == null) break;
    CompletableFuture<Map<String,Integer>> future = CompletableFuture.sup
```

```
plyAsync(() -> {
        Map<String,Integer> localCounts = new HashMap<>();
        Iterable<String> words = new Words(page.getText());
        for (String word : words) {
            if (word.length() > 1 || word.equals("a") || word.equals("I")
) {
                localCounts.merge(word, 1, Integer::sum);
            }
        }
        return localCounts;
    }, executor);
    futures.add(future);
    processedPages++;
}
```

- Each page is processed asynchronously using `CompletableFuture.supplyAsync()`.
- Each task extracts words and creates a local word frequency map
- The tasks are submitted to the thread pool and their futures are collected

Results Aggregation

```
CompletableFuture<Void> allDone = CompletableFuture.allOf(futures.toArray
(new CompletableFuture[0]));
allDone.join();

Map<String,Integer> combinedCounts = new HashMap<>();
for (CompletableFuture<Map<String,Integer>> f : futures) {
    f.join().forEach((k, v) -> combinedCounts.merge(k, v, Integer::sum));
}
```

- `CompletableFuture.allOf()` waits for all tasks to complete.
- The join() method blocks until all tasks are done.
- The results from each future are merged into a single map containing the total word counts.

Results Processing

```
LinkedHashMap<String,Integer> commonWords = new LinkedHashMap<>();
combinedCounts.entrySet().stream()
        .sorted(Map.Entry.comparingByValue(Comparator.reverseOrder()))
        .forEachOrdered(e -> commonWords.put(e.getKey(), e.getValue()));

commonWords.entrySet().stream().limit(3).toList()
        .forEach(e -> System.out.println("Word: '" + e.getKey() + "' with
total " + e.getValue() + " occurrences!"));
```

- The final word counts are sorted and displayed.

Garbage Collector Tuning

Overview

To further optimize the performance of each implementation, we conducted a series of garbage collection (GC) benchmarks using different Java GC algorithms and heap configurations. The primary goal was to evaluate how different collectors affect memory management, pause times, and overall execution time when processing large volumes of text.

Testing Setup

We designed a batch script to automate testing across multiple configurations. For each test, the script:

- Sets heap size (`-Xms`, `-Xmx`)
- Selects the GC via `-XX:+Use[GC]`
- Enables verbose GC logging (`-Xlog:gc*`)
- Executes the target implementation (e.g., `Sequential`, `Multithreaded`)
- Stores logs and console output in timestamped directories under `GarbageLogs`

Garbage Collectors Tested

GC Algorithm	JVM Flag	Notes
Serial GC	`-XX:+UseSerialGC`	Simpler, single-threaded; suitable for small heaps
Parallel GC	`-XX:+UseParallelGC`	Throughput-optimized; allows tuning pause times
G1 GC	`-XX:+UseG1GC`	Balanced collector; supports `MaxGCPauseMillis`
ZGC	`-XX:+UseZGC`	Low-latency, scalable; suited for large heaps

Test Matrix

Tests were run with varying heap configurations:

- `128m–1g`, `256m–2g`, `512m–4g`, `1024m–8g`
- Additional tuning for:
    - `MaxGCPauseMillis` (e.g., 25, 50, 100, 200 ms)
    - `ParallelGCThreads`, `ConcGCThreads`

- `G1HeapRegionSize`

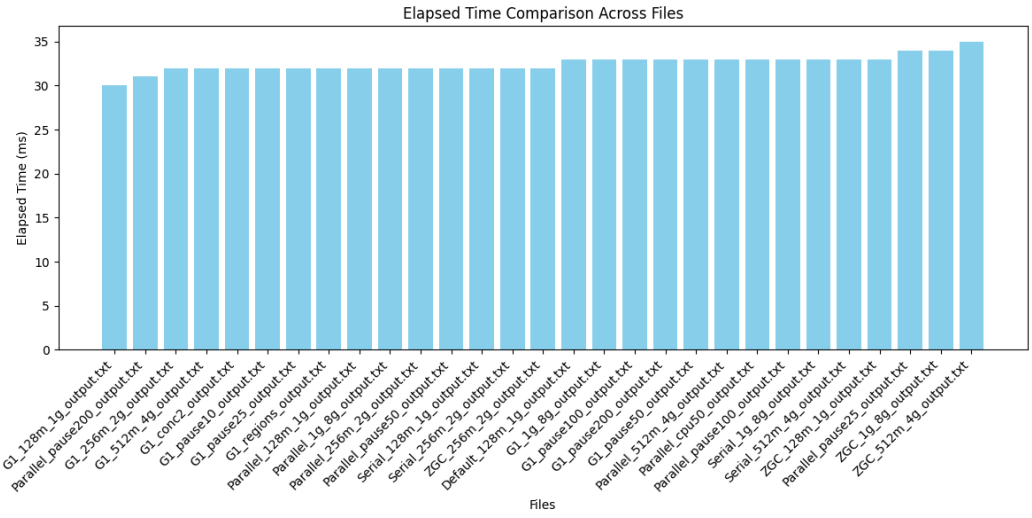
Each combination was executed against the same input file (`enwiki.xml`) and the same number of pages (`maxPages = 100000`) to ensure comparability.

Results Analysis

Sequential Implementation

The one in this implementation with the best elapsed time was the G1_128_1G. Using the flowing parameters:

- G1 Garbage Colector
- Initial memory of 128 MB
- Grows up to 1G if needed

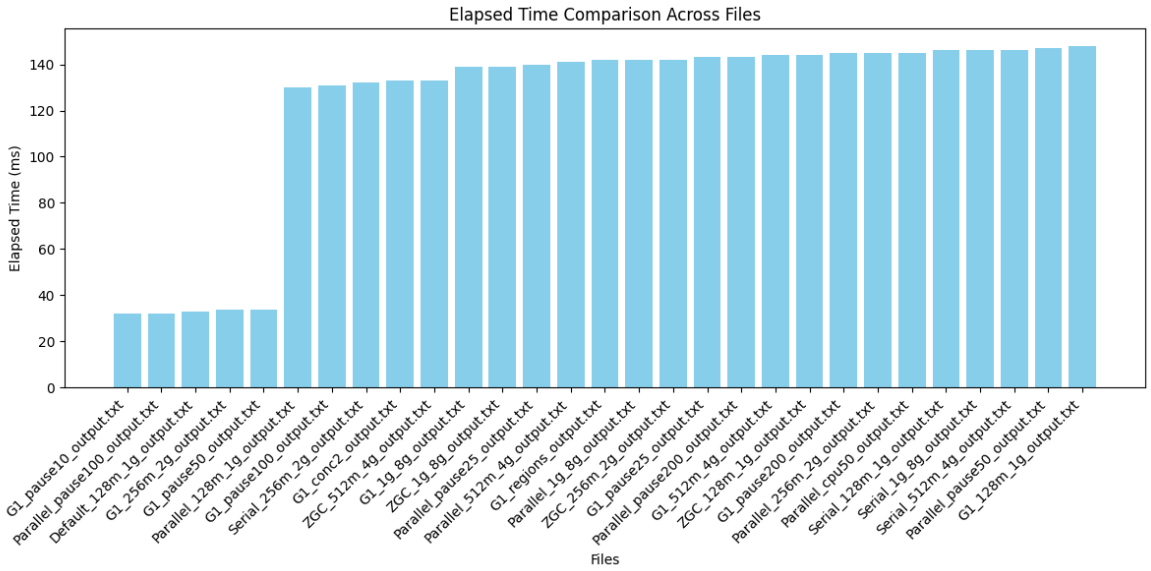


Multithreaded Implementation (No Thread Pool)

The one in this implementation with the best elapsed time was the G1_128_1G. Using the flowing parameters:

- G1 Garbage Colector
- Initial memory of 256MB
- Grows up to 2G if needed

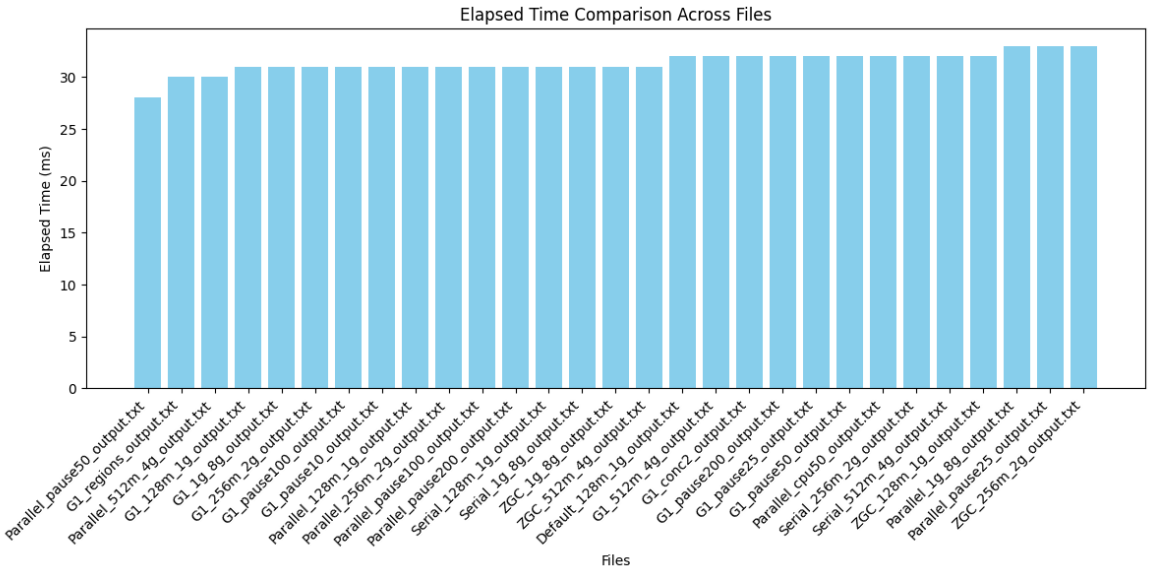
- A target GC pause of 10ms



Multithreaded Implementation (Thread Pool)

The one in this implementation with the best elapsed time was the G1_128_1G. Using the flowing parameters:

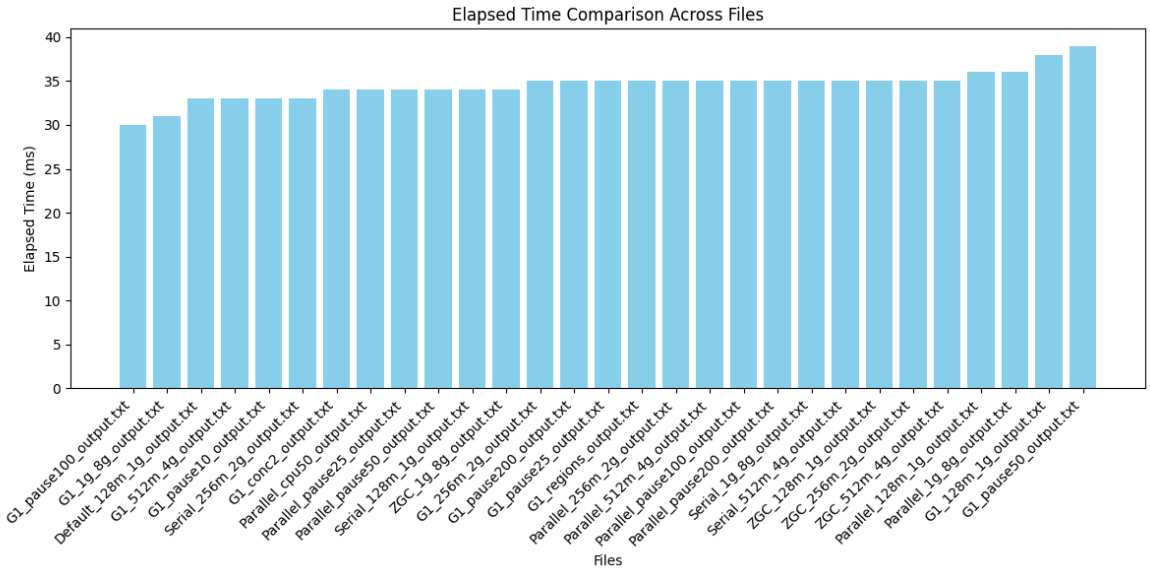
- A target GC pause of 50 ms
- Initial memory of 256MB
- Grows up to 2G if needed
- G1 Garbage Colector



Fork/Join Implementation

The one in this implementation with the best elapsed time was the G1_128_1G. Using the flowing parameters:

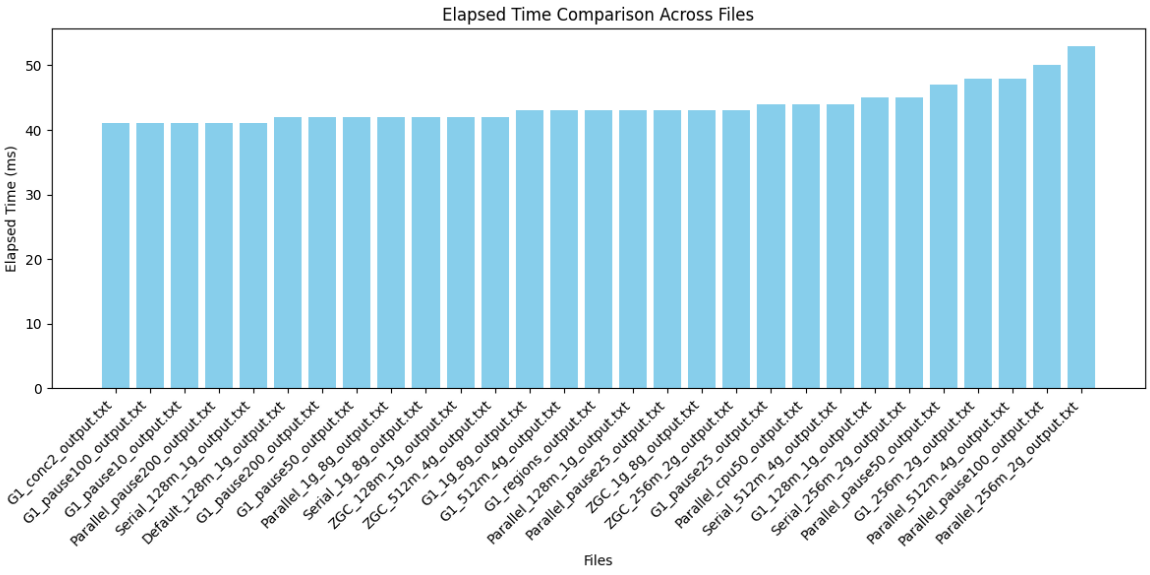
- G1 Garbage Colector
- A target GC pause of 100 ms
- Initial memory of 256MB
- Grows up to 2G if needed



CompletableFuture

The one in this implementation with the best elapsed time was the G1_con2. Using the flowing parameters:

- G1 Garbage Colector
- Sets the number of threads G1 uses for concurrent phases like marking and cleanup.
- Initial memory of 256MB
- Grows up to 2G if needed



Concurrency and Synchronization

Overview

This section analyzes the concurrency models and synchronization mechanisms employed across our different implementations, highlighting the strengths, weaknesses, and design considerations of each approach.

Sequential Implementation

The sequential implementation serves as our baseline and involves no concurrency or synchronization mechanisms:

- **Concurrency Model**: None of the program processes pages one at a time.

- **Synchronization Mechanisms**: Not applicable, no shared resources or threads to manage.

- **Thread Safety**: Inherently thread-safe due to single-threaded execution

- **Data Structures**: Uses a simple `HashMap` for word counting, while this implementation avoids synchronization overhead, it cannot leverage multiple CPU cores, resulting in suboptimal performance on modern multicore systems.

Multithreaded Implementation (No Thread Pool)

This implementation employs explicit thread management with a producer-consumer pattern:

- **Concurrency Model**: Multiple manually created and managed threads

- **Synchronization Mechanisms:**

    - **BlockingQueue** for thread-safe data transfer between producer and consumer threads.

    - **AtomicBoolean** to signal when the producer has finished processing.

- **ConcurrentHashMap** for thread-safe word counting.

- **Thread Safety**: The use of `BlockingQueue` and `ConcurrentHashMap` ensures thread safety during concurrent access.

- **Data Structures**: `BlockingQueue` for page transfer, `ConcurrentHashMap` for word counting.

The producer-consumer design effectively decouples page reading from processing, allowing these operations to proceed in parallel, however, this implementation faces several synchronization challenges:

- **Thread Management**: Manual thread creation and management can lead to overhead and complexity.
- **Termination Detection**: Consumers must check both queue emptiness and producer completion flag to determine when to terminate.
- **Thread Management Overhead**: Manual thread creation and lifecycle management adds complexity

Multithreaded Implementation (Thread Pool)

The thread pool implementation improves upon the explicit threading model:

- **Concurrency Model:** Fixed-size thread pool managed by ExecutorService.

- **Synchronization Mechanisms:**

    - **ExecutorService** for managing thread lifecycles.

    - **ConcurrentHashMap** for thread-safe word counting.

- **Thread Safety:** The use of `ConcurrentHashMap` ensures thread safety during concurrent access.

- **Data Structures:** `ExecutorService` for thread management, `ConcurrentHashMap` for word counting.

This implementation addresses the thread management overhead of the previous approach by delegating thread lifecycle management to the ExecutorService, key synchronization aspects include:

- **Task Submission:** Main thread submits page processing tasks to the executor

- **Thread reuse:** Instead of creating new threads for each page, the pool reuses a fixed number of worker threads

- **Termination Handling:** awaitTermination() provides clean synchronization for completion detection

The thread pool efficiently manages thread resources, preventing the overhead of creating and destroying threads for each task, which can be significant when processing many small tasks.

Fork/Join

The Fork/Join implementation uses a divide-and-conquer approach:

- **Concurrency Model:** Work-stealing thread pool specialized for recursive decomposition.
- **Synchronization Mechanisms:**
    - **ForkJoinPool** for managing thread lifecycles and work-stealing.
    - **RecursiveTask** for defining tasks that can be split into subtasks.
    - **ConcurrentHashMap** for thread-safe word counting.
- **Thread Safety:** The use of `ConcurrentHashMap` ensures thread safety during concurrent access.
- **Data Structures:** `ForkJoinPool` for thread management, `ConcurrentHashMap` for word counting.

This implementation employs the following synchronization strategy:

- **Task Splitting:** Pages are recursively split until reaching a threshold size.

- **Local Processing:** Each subtask maintains its own HashMap, avoiding synchronization during the computation phase.

- **Merging Results:** The mergeMaps() method combines results from subtasks, using `ConcurrentHashMap.merge()` for thread-safe updates.

- **Work-Stealing:** The ForkJoinPool automatically balances workload across threads.

This approach minimizes synchronization overhead by, eliminating shared mutable state during the computation phase, deferring synchronization to the join phase and using work-stealing to dynamically balance thread workloads

The threshold value significantly impacts performance by controlling the granularity of parallelism - too small creates excessive overhead, while too large underutilizes available cores.

CompletableFuture

The CompletableFuture implementation represents a more modern approach to concurrency:

- **Concurrency Model:** Asynchronous task execution with callbacks.

- **Synchronization Mechanisms:**

- **ExecutorService** for managing thread lifecycles.

- **CompletableFuture** for asynchronous task execution and result handling.

- **ConcurrentHashMap** for thread-safe word counting.

- **Thread Safety:** The use of `ConcurrentHashMap` ensures thread safety during concurrent access.

- **Data Structures:** `ExecutorService` for thread management, `CompletableFuture` for asynchronous task handling, `ConcurrentHashMap` for word counting.

This implementation features sophisticated synchronization patterns:

- **Isolated State:** Each task operates on its own local HashMap, eliminating write contention.

- **Future Composition:** CompletableFuture.allOf() coordinates the completion of all tasks.

- **Explicit Join:** The join() method provides synchronization points for result aggregation.

The key advantage is the clean separation between task submission and result aggregation, with an explicit synchronization point using CompletableFuture.allOf().join() , this approach, reduces synchronization overhead during parallel execution, simplifies error handling and task coordination and provides a more declarative programming model

Overall Comparison

Implementation	Shared Mutable State	Synchronization Mechanism	Contention Points	Synchronization Overhead
Sequential	None	None	None	None
Multithreaded (No Pool)	BlockingQueue, ConcurrentHashMap	Blocking operations, atomic operations	Queue access, map updates	Medium-High
Thread Pool	ConcurrentHashMap	Thread pool internal synchronization	Map updates	Medium

Implementation	Shared Mutable State	Synchronization Mechanism	Contention Points	Synchronization Overhead
Fork/Join	None during computation	Fork and join operations	Result merging	Low
CompletableFuture	None during computation	Future composition	Result aggregation	Low

Performance Analysis

Sequential

As the volume of data increases, the execution time grows exponentially, demonstrating no ability to leverage multiple cores. There is no mechanism to distribute the workload or to parallelize operations, which severely limits the system's ability to scale efficiently. Since all processing is performed by a single execution thread, the overall response time becomes significantly higher, which is particularly critical in real-time or large-scale data processing scenarios. While the main core is fully utilized, the remaining cores remain idle, leading to an inefficient use of the system's computational potential.

The CPU time was particularly high, reaching values close to 30,000 milliseconds for the largest datasets, which confirms that a single thread was solely responsible for all computation without any parallel acceleration. In terms of **s**peedup, the sequential baseline remained at 1, with no improvement even when more threads were available, and the parallel efficiency was effectively zero. Although memory usage was low, the lack of concurrency made the solution unsuitable for scalable, high-performance applications.

Multithreaded Implementation (No Thread Pool)

In the multithreaded implementation without the use of thread pools, multiple threads are explicitly created and managed to process the data concurrently. The implementation exhibits a relatively consistent pattern across different thread counts, with execution times showing only moderate variation as threads increase from 1 to 500. For smaller file sizes (10,000 pages), the execution time remains close

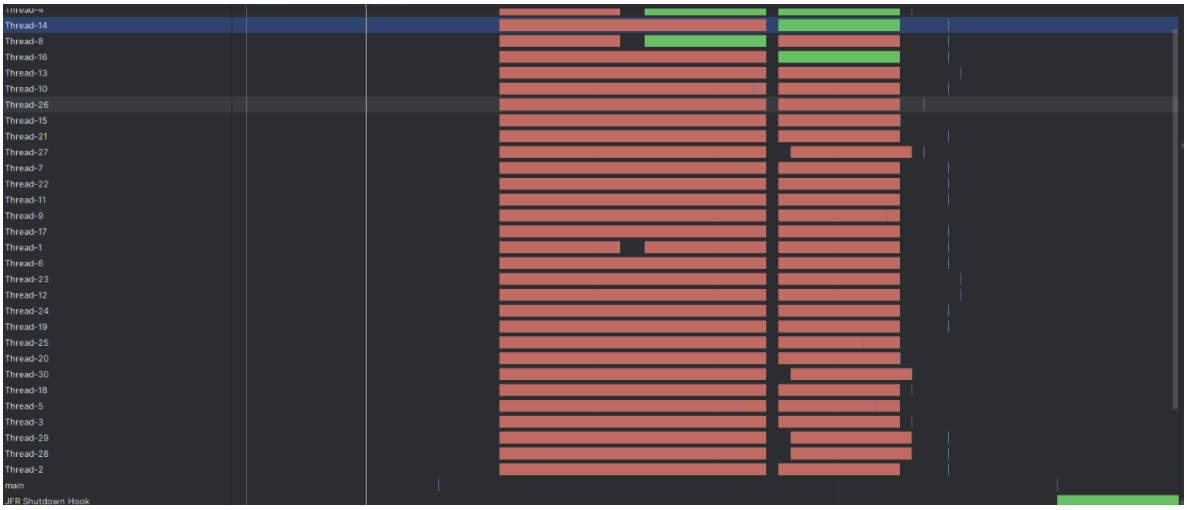
to 1,500–1,800 ms, indicating efficient utilization of threads without significant overhead. As the number of pages increases to 50,000 and 100,000, execution time rises accordingly, reaching approximately 4,000 ms and 7,000 ms, respectively. This growth is expected due to the higher workload.

Notably, increasing the number of threads beyond a certain point (e.g., 100 threads) does not yield substantial performance gains. In fact, for larger file sizes, the execution time slightly increases with more threads. This suggests that the overhead of managing a higher number of threads starts to outweigh the parallelization benefits. The relatively flat slope across thread counts for each file size supports this observation, highlighting diminishing returns as thread count grows.

In relation to CPU time, the implementation exhibits a notable increase as the number of threads increases. For small datasets (10,000 pages), CPU time starts low—close to 0 ms at 50 threads—but rises significantly at higher thread counts. This pattern is consistent for 50,000 and 100,000 pages, with CPU time increasing from a few hundred milliseconds at 50 threads to over 1,000 ms and 1,800 ms respectively at 500 threads.

This behavior suggests that while wall-clock execution time remains relatively stable, thread scheduling and context-switching overhead grow with the number of threads, consuming more CPU resources. It indicates suboptimal thread management, where increasing concurrency does not translate to proportional performance gains and instead leads to higher CPU load.

In the flowing image we can see the display of multiple threads running simultaneously, only a few display a green block suggesting active computation is not evenly distributed, leading to a suboptimal performance for most workloads.



Multithreaded Implementation (Thread Pool)

The Multithreaded Thread Pools implementation shows stable and generally favorable performance across varying thread counts and dataset sizes. Execution time scales predictably with the size of the input: approximately 1,500 ms for 10,000 pages, around 4,000 ms for 50,000 pages, and just under 7,000 ms for 100,000 pages. These results are consistent and indicate that the thread pool-based approach handles increasing workloads efficiently.

Importantly, the variation in execution time across different thread counts (50, 100, 500) is minimal for all dataset sizes. The stable performance indicates a well-balanced distribution of tasks among threads and low overhead in thread lifecycle management.

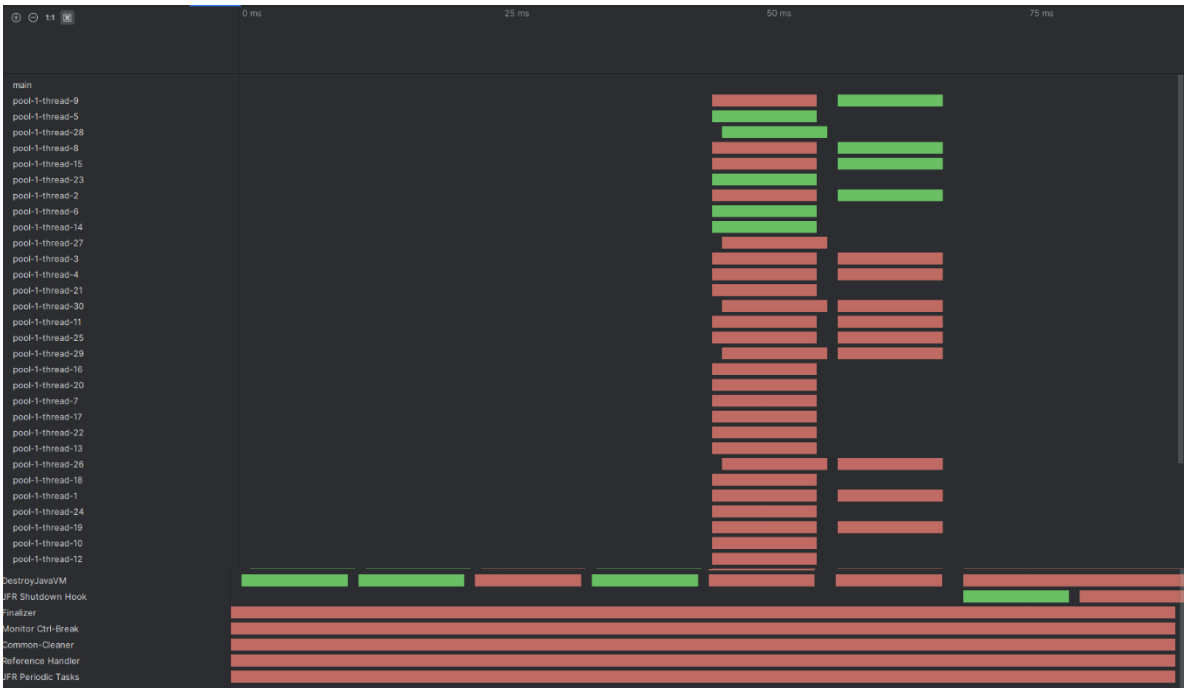
Compared to other multithreaded implementations, this approach provides a good trade-off between performance and scalability. While it does not exhibit significant performance gains when increasing the number of threads beyond 50, it also does not suffer from performance degradation. This is advantageous in environments where thread resources are constrained or when predictable performance is preferred over aggressive parallel scaling.

Regarding CPU Time, the thread pool maintained a balanced load across all cores, achieving moderate CPU usage that grew proportionally with the dataset size but without excessive overhead. This indicates efficient parallel computation without unnecessary context switching or idle times.

In terms of Memory Usage, the thread pool implementation was among the most efficient. Memory consumption remained stable and relatively low, avoiding the spikes seen in manual thread creation models. By reusing a fixed number of threads, it minimized allocation and deallocation costs.

The Speedup graphs demonstrated that the thread pool achieved significant improvements over the Multithreaded Implementation without thread pools. Speedup increased quickly with the number of threads up to a point (around 50–100 threads), after which gains plateaued, reflecting the natural saturation point of available hardware resources.

The flowing image demonstrates a more mature concurrency approach than raw multithreading, with better control over thread lifecycle and improved resource utilization. It balances the parallelism benefits of multithreading with the efficiency of resource reuse, making it suitable for server applications and tasks requiring controlled concurrency.



Fork/Join

The Fork/Join implementation demonstrated important concurrency capabilities by recursively decomposing the workload into smaller, independent subtasks. This allowed it to efficiently utilize multiple cores, leading to performance improvements when compared to the sequential baseline. Unlike the sequential approach, which suffered from linear scalability limitations and single-core utilization, Fork/Join attempted to distribute the workload and take advantage of parallelism, particularly on large datasets.

Overall, the solution shows a mixed and less consistent execution time profile compared to other multithreaded approaches. Across all tested page sizes—10,000, 50,000, and 100,000 pages—the execution time generally increases as the number of threads increases. This is particularly evident in the case of 100,000 pages, where the execution time rises from approximately 14,000 ms at 50 threads to around 15,000 ms at 500 threads. This upward trend is contrary to the expected behavior of a parallel processing framework and suggests inefficiencies in thread management or task decomposition.

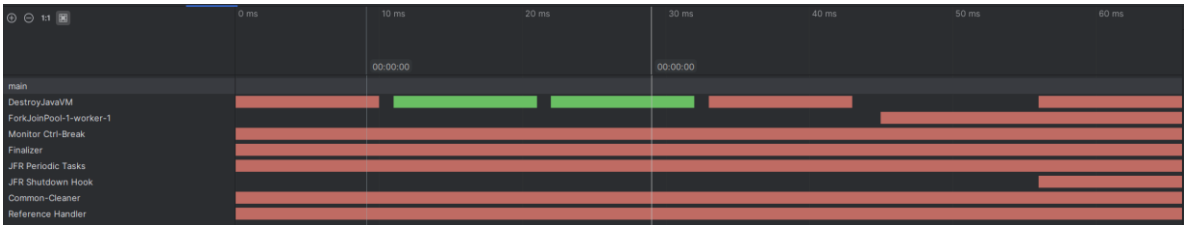
For smaller inputs, such as 10,000 pages, the execution time initially improves when moving from 1 to 50 threads but then gradually increases again with more threads. This indicates that while the Fork/Join framework may provide benefits up to a certain

degree of parallelism, it suffers from increased overhead as thread count rises—likely due to the cost of managing a large number of tasks and worker threads, or because of imbalanced workload distribution among threads.

In relation to CPU time, it shows a relatively stable time across all thread counts, especially for the 50,000 and 100,000-page datasets. For 10,000 pages, CPU time remains around 1,500 ms and increases modestly with more threads, reaching just under 2,000 ms at 500 threads. For larger datasets, the CPU time remains consistent between 3,000–5,000 ms regardless of thread count.

This suggests that the Fork/Join framework distributes tasks efficiently across threads without excessive overhead. The controlled and bounded increase in CPU time indicates balanced load distribution and effective use of worker threads. It demonstrates good resource utilization, especially in comparison to the steeper increases observed in the basic multithreaded implementation.

In the flowing image we can see this timeline is cleaner, compared to the other approaches, there is a dedicated “ForkJoinPool-1-worker-1", representing the main worker in the Fork/Join framework, there are fewer but longer suggesting more substantial units of work being processed.
There's less thread overhead compared to raw multithreading or even standard thread pools, displaying to appear more coordinated compared to the other implementations.



CompletableFuture

The Completable Future Solution displays moderately consistent performance across varying thread counts, though it does not scale as efficiently as expected for parallel execution. For all tested file sizes—10,000, 50,000, and 100,000 pages—the execution time decreases noticeably from 1 to 50 threads, suggesting initial gains from parallel task submission and execution through the Completable Future framework. However, from 50 to 500 threads, the performance plateaus or even degrades slightly, particularly for the largest file size (100,000 pages), where the execution time levels off around 14,000 ms.

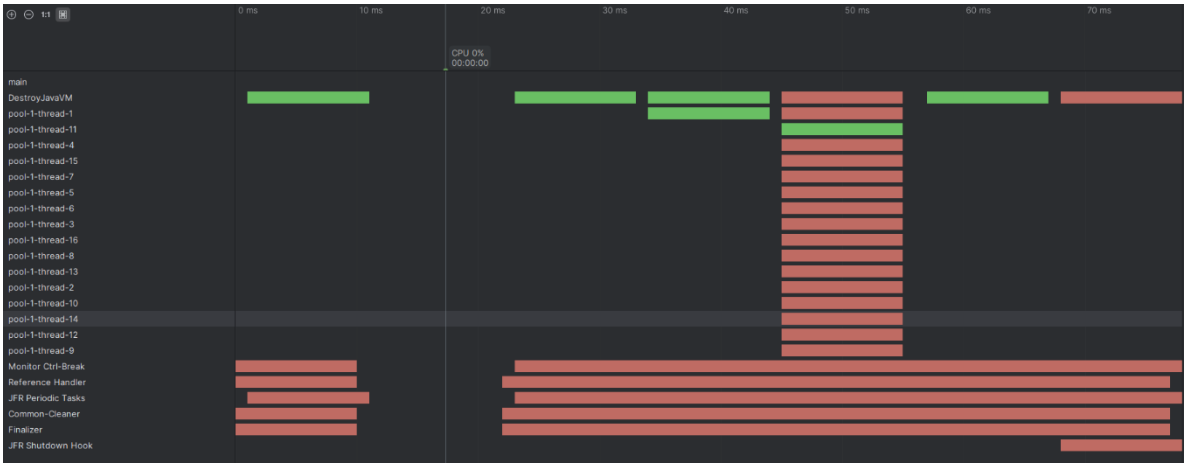
For smaller datasets, such as 10,000 and 50,000 pages, the execution times remain in the range of approximately 2,000 ms to 7,000 ms. These results indicate a reasonable level of concurrency, but the limited improvement beyond 50 threads suggests that the implementation may not be effectively utilizing additional threads. This could stem from a fixed-size thread pool, overhead in task orchestration, or limitations in non-blocking task completion management.

Notably, the gap between execution times for 50,000 and 100,000 pages is larger than between 10,000 and 50,000 pages, implying that the implementation does not scale linearly with input size. This could be due to bottlenecks in asynchronous coordination or inefficient division of work among futures, particularly when handling a large number of subtasks.

The solution shows moderate CPU usage that scales consistently with input size but remains mostly flat across different thread counts. For 10,000 pages, CPU time stays close to 1,500 ms across all thread levels. For 50,000 and 100,000 pages, CPU time ranges between 3,000 ms and 6,500 ms, with only minor variation as the number of threads increases from 50 to 500.

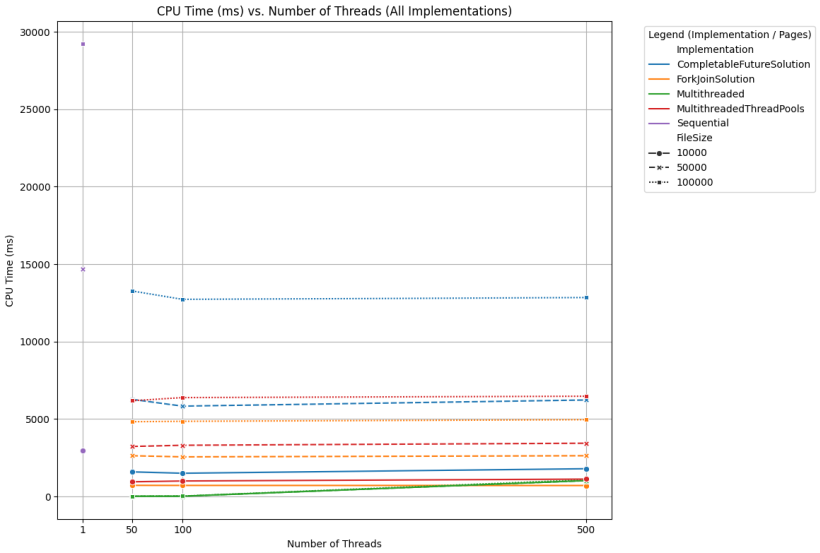
This pattern indicates efficient use of asynchronous computation without excessive thread scheduling overhead. It reflects a good balance between concurrency and system resource usage.

The flowing image displays the Completable Future solution demonstrating a modern, flexible concurrency approach ideal for asynchronous, non-blocking operations. Its ability to compose, combine and chain asynchronous tasks creates a responsive execution pattern suitable for I/O-bound and event-driven applications. This approach excels at managing dependencies between asynchronous operations while efficiently utilizing thread resources, making it particularly valuable for responsive applications and complex asynchronous workflows.



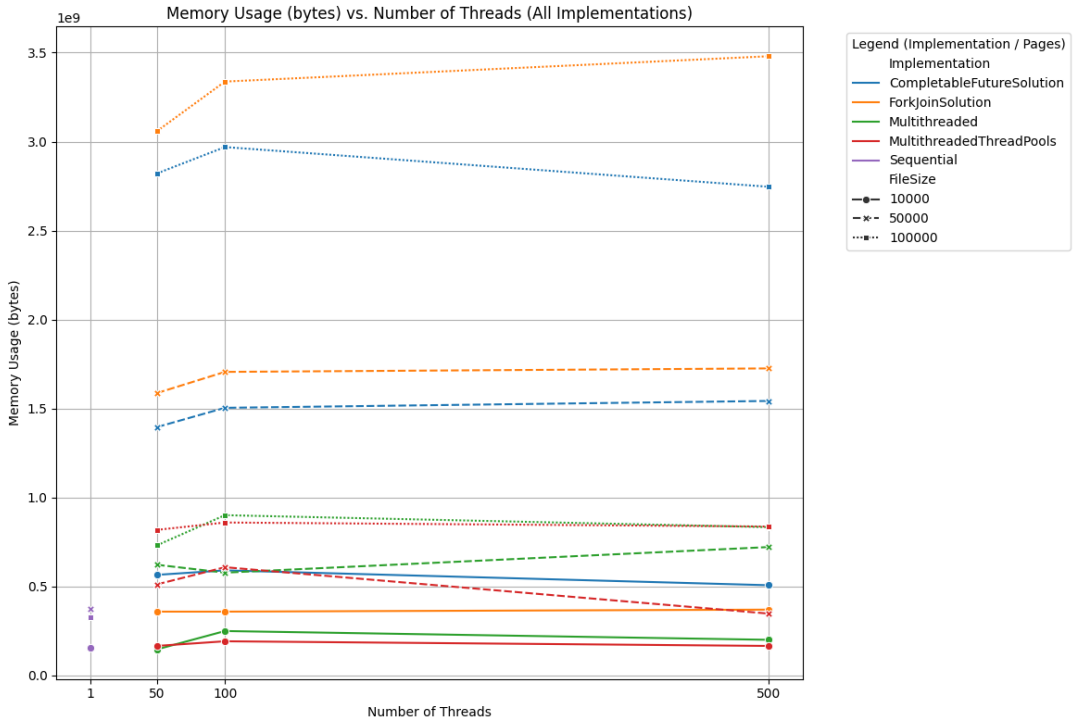
Overall CPU Analysis

The graph reveals that while the sequential implementation has the highest CPU time, the multithreaded approaches show varying degrees of efficiency across thread counts, with the basic multithreaded implementation performing best at lower thread counts but experiencing degradation as threads increase, while CompletableFuture and ThreadPool implementations maintain more consistent performance regardless of thread scaling.



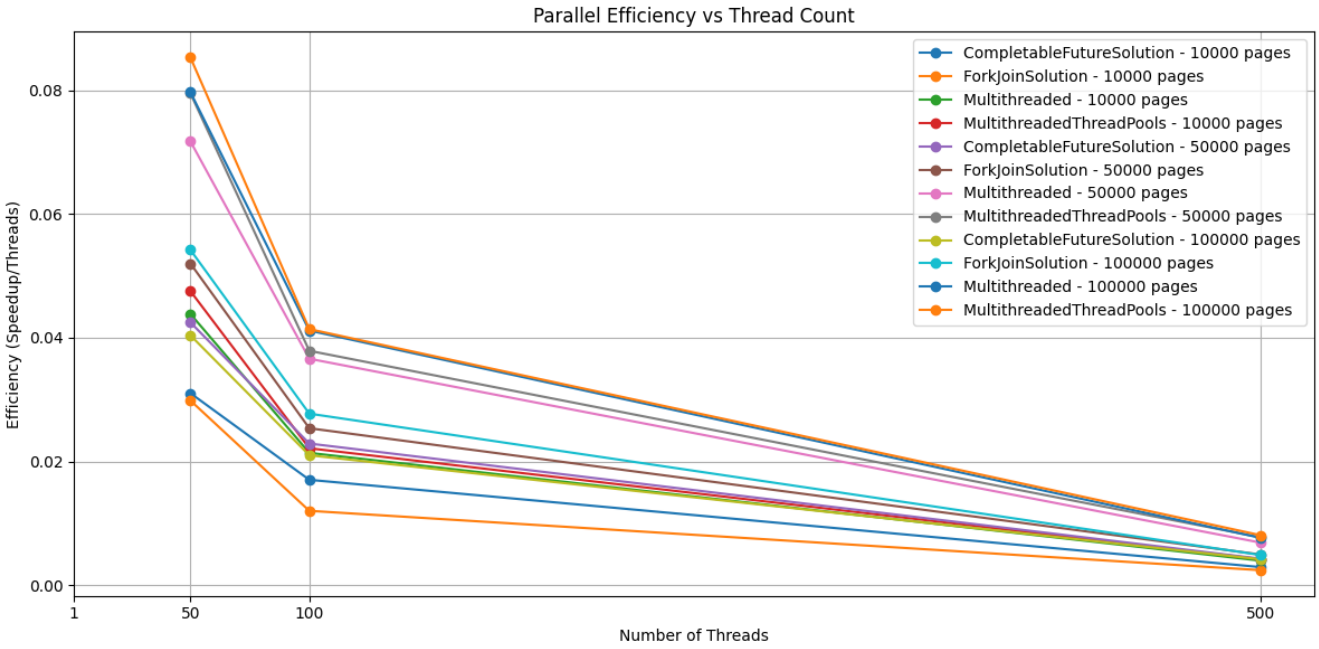
Overall Memory Efficiency Analysis

The memory usage graph reveals that ForkJoinSolution consumes significantly more memory than other implementations (especially with 100,000 pages), while CompletableFutureSolution shows high but decreasing memory usage as threads increase, and both Multithreaded and MultithreadedThreadPools maintain the lowest and most stable memory footprints across different thread counts, with the Sequential implementation using minimal memory at low thread counts but not scaling to higher thread configurations.



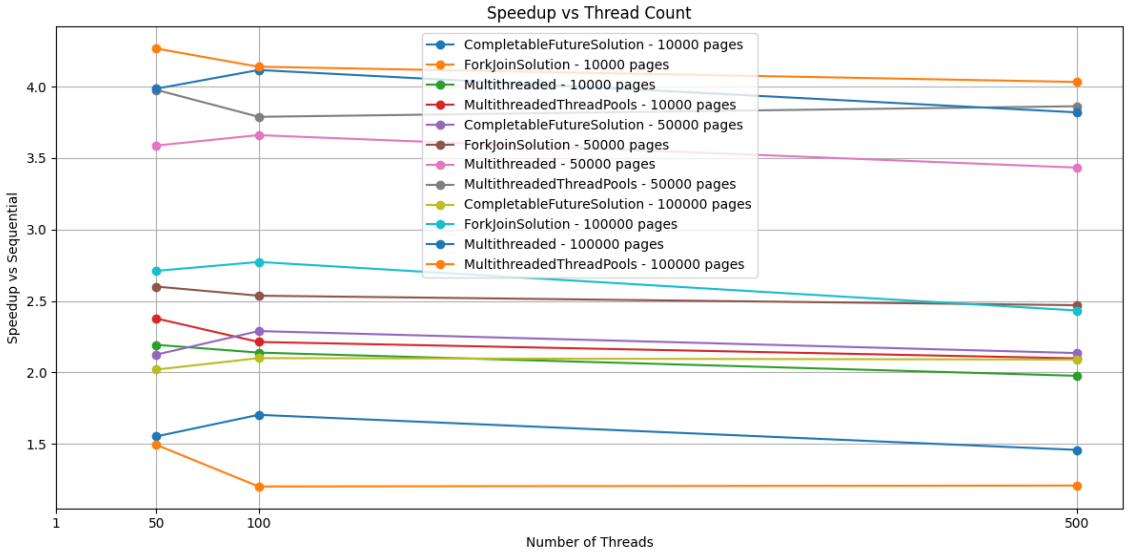
Overall Parallel Processing Analysis

The graph illustrates a universal decline in parallel efficiency (speedup/threads) across all implementations as thread count increases from 50 to 500, with MultithreadedThreadPools and ForkJoinSolution showing the highest initial efficiency at 50 threads for larger datasets, but all implementations converging to similarly low efficiency levels at 500 threads, demonstrating the classic parallel computing challenge of diminishing returns with increased parallelization.



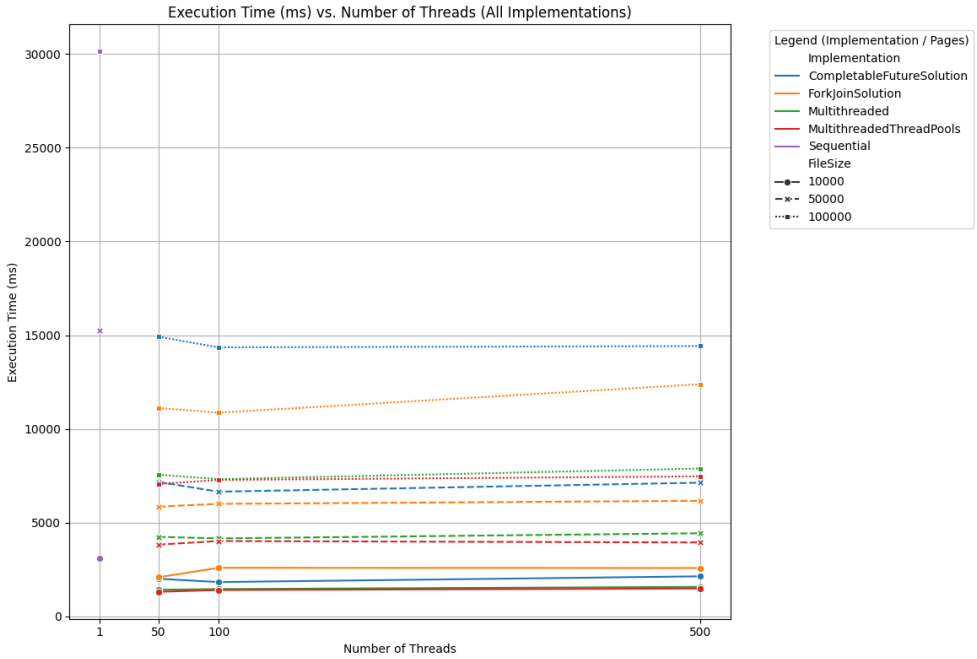
Overall Speedup Comparison

The graph shows that ForkJoinSolution and MultithreadedThreadPools achieve the highest speedups relative to sequential processing (reaching 4x for smaller datasets), while most implementations demonstrate minimal speedup improvement or even slight degradation when thread count increases beyond 100, with performance characteristics being largely consistent across thread counts but varying significantly based on implementation type and dataset size.



Overall Elapsed Time Comparison

The graph reveals that sequential implementation has dramatically higher execution times (30,000ms) than parallel approaches, with CompletableFutureSolution showing stable but high execution times across thread counts for large datasets, while MultithreadedThreadPools consistently delivers the best performance across file sizes and thread counts, and most implementations show negligible execution time improvements beyond 50 threads, indicating that increased thread counts provide minimal benefit beyond certain thresholds.



Conclusion

This project implemented and analysed five approaches for processing large volumes of textual data from Wikipedia: Sequential, Multithreaded (No Thread Pool), Multithreaded (Thread Pool), Fork/Join, and CompletableFuture. Our analysis revealed significant insights into their performance characteristics and applicability.

The **sequential approach** served as a reliable baseline but highlighted the limitations of single-threaded processing on modern multicore systems. The **manual multithreaded implementation**, despite enabling concurrency, introduced significant overhead due to manual thread management and complex synchronization requirements, often performing worse than the sequential model under heavier loads.

In contrast, the **thread pool implementation** demonstrated substantial performance improvements by efficiently reusing threads and minimizing overhead. It

struck a strong balance between simplicity, efficiency, and scalability, making it the most robust and performant solution overall. The **Fork/Join framework** effectively decomposed tasks into subtasks and leveraged work-stealing, offering good CPU utilization and scalability, though its recursive structure introduced task management overhead that limited performance on smaller datasets.

Finally, the **CompletableFuture model** showcased a modern, asynchronous paradigm well-suited for tasks involving complex dependencies and non-blocking execution. It offered a clean, declarative style and competitive performance, especially in scenarios involving asynchronous workflows or I/O-bound tasks.

Additionally, the inclusion of **Garbage Collector tuning and analysis** added a valuable dimension to the performance assessment, reinforcing the impact of JVM configuration on application efficiency.

Best and Worst Implementations

Best Implementation: Multithreaded (Thread Pool) The Thread Pool implementation consistently emerged as the most effective solution across our test scenarios. It delivered the optimal balance of performance, resource utilization, and implementation complexity for our word frequency analysis task. Key advantages include:

- Most consistent execution times across varying thread counts (stable performance from 50-500 threads)
- Lowest and most stable memory footprint among parallel implementations
- Excellent CPU utilization without excessive context switching
- Simplified thread lifecycle management compared to manual approaches
- Best overall speedup-to-resource-consumption ratio

Worst Implementation: Sequential The Sequential implementation proved to be the least effective approach, particularly for larger datasets. Despite its simplicity, it suffered from:

- Dramatically higher execution times (reaching 30,000ms for large datasets compared to ~7,000ms for Thread Pool)
- Inability to utilize available CPU cores (single-threaded execution)
- Linear scaling with input size, making it impractical for production workloads
- Zero parallel efficiency and speedup regardless of system capabilities
- Complete breakdown in performance with increasing dataset sizes